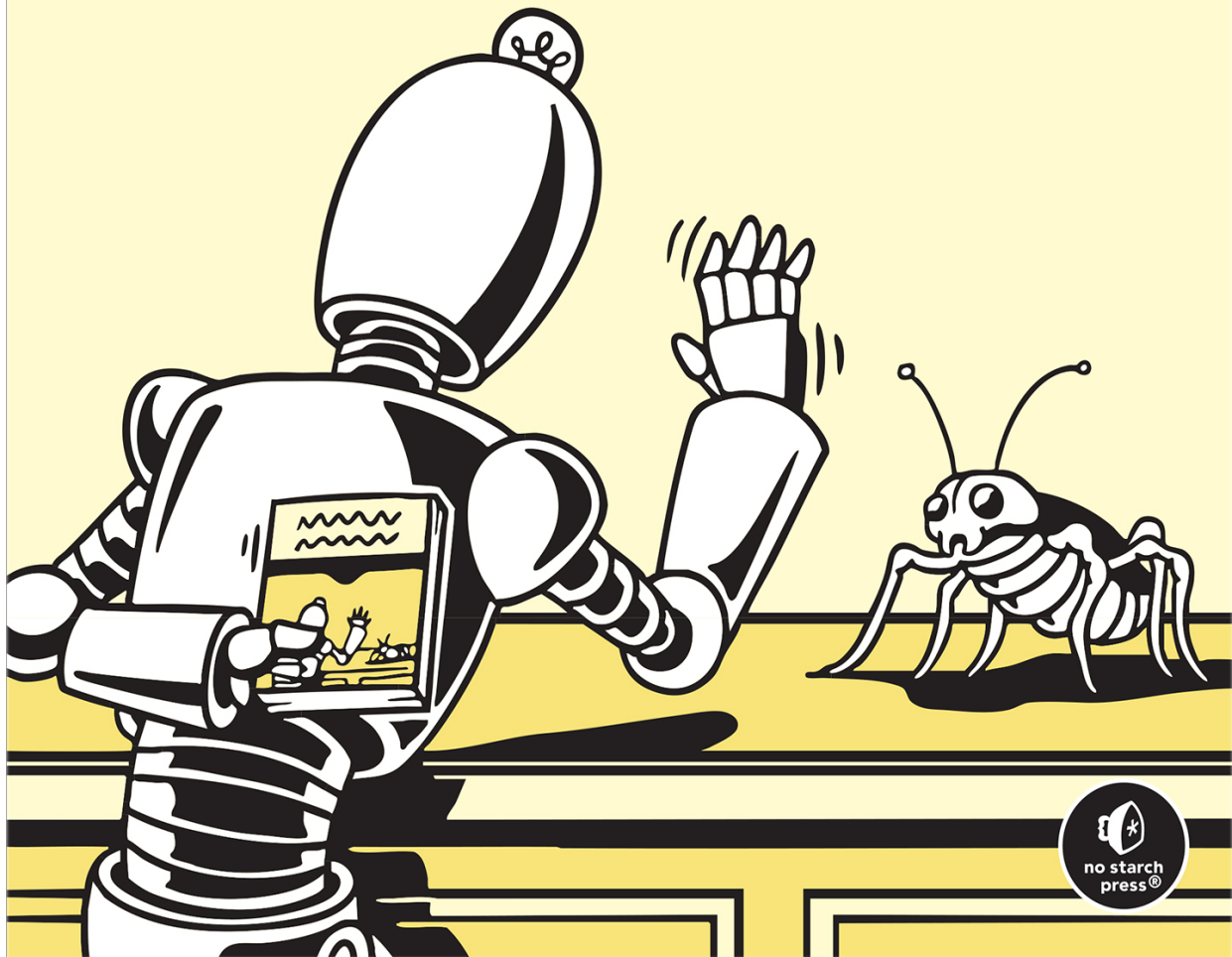


THE BOOK OF DEBUGGING

A SYSTEMATIC WORKFLOW FOR
FINDING AND FIXING BUGS

JOHANNES KUHLMANN



CONTENTS IN DETAIL

[TITLE PAGE](#)

[COPYRIGHT](#)

[ABOUT THE AUTHOR AND THE TECHNICAL REVIEWER](#)

[ACKNOWLEDGMENTS](#)

[INTRODUCTION](#)

[What Is a Bug?](#)

[Why We Should Fix Bugs](#)

[Debugging Is Inevitable](#)

[Debugging Is Your Job](#)

[Who This Book Is For](#)

[Prerequisites](#)

[How to Use This Book](#)

[Manage Expectations](#)

[THE WORKFLOW](#)

[Step 1: Reproduce](#)

[Step 2: Probe](#)

[Step 3: Examine](#)

[Step 4: Fix](#)

[Handle Bugs as a Team](#)

[Common Root Causes](#)

[Bad Data](#)

[Bad Pointers](#)

[Bad Logic](#)

[Edge Cases](#)

[Floating-Point Inaccuracies](#)

[Stale Code or Assets](#)

[Wrong Timing or Order](#)

[Unclear or Bad Requirements](#)

[User Experience Issues](#)

[Summary](#)

PART I: REPRODUCE

What Makes a Great Bug Report?

1. Reproduce the Bug Yourself

Re-create the User Experience

Emulate the Reporter

Introduce Variations

Beware of Networking Complexity.

2. Shorten the Feedback Loop

Reduce Build Times

Reduce Deployment Times

Reduce Startup Times

Jump to the Correct Location

3. Automate the Reproduction

Whether to Automate

How to Automate

When You Can't Automate

Summary.

PART II: PROBE

4. Try Shortcuts

Make Educated Guesses

Focus on Suspicious Systems

Ignore Irrelevant Systems

Know When to Change Your Strategy.

5. Search the Internet

Don't Scroll Too Far

Try Different Search Engines

Use Advanced Search Engine Features

Search for Source Code

Don't Forget Your Intranet

6. Check Version Histories

Find a Working Version

Decide What to Test

Identify the Offending Change

Look into the Future

Look into Branches and Forks

7. Attach the Debugger by Default

Inspect Your Program's Behavior

Use Exception Handling

Use Breakpoints to Inspect Your Code's Execution

Set Up Debuggers on Difficult Platforms

Handle Cases Where the Debugger Interferes

8. Instrument the Program to Report Problems

Use Assertions

Handle Errors

Signal Errors to the Caller

Use Obvious Test Patterns

[Don't Break the Reproduction with Modifications](#)
[Make the Instrumentation Optional](#)

9. Remove Randomness

[Create Deterministic Programs](#)
[Disable Potential Sources of Randomness](#)
[Put Determinism to Work](#)
[Consider Rare Hardware Issues](#)

10. Ask Colleagues for Help

[Ask Your Team](#)
[Ask for External Support](#)

11. Search in All Files

[Choose and Configure Your Search Tool](#)
[Choose Useful Search Terms](#)
[Search Beyond the Source Code](#)
[Peek into Nontext Files](#)
[Look for Strings Composed in Code](#)
[Use Searchable Patterns in Code](#)

12. Collect Crash Dumps

[Trigger a Crash Dump](#)
[Get Information from a Crash Dump](#)
[Implement Crash Dumps](#)
[Turn Crash Dumps into Actionable Data](#)
[Use Third-Party Tools to Simplify Crash Reporting](#)

[Summary](#)

PART III: EXAMINE

13. Step Through the Code

[Choose the Right Build Configuration](#)
[Start the Debugging Process](#)
[Use Advanced Debugging Features](#)

14. Read the Code

[Act Like a Virtual Machine](#)
[Strive for Readable Code](#)
[Record What You Find](#)
[Know Your Programming Languages](#)

15. Follow the Data

[Find the Original Data](#)
[Spot the Symptoms of Bad Data](#)
[Find Where the Data Becomes Bad](#)
[Compare Good and Bad Data](#)
[Follow Good Data](#)

16. Question Everything

[Find the Cause That Must Exist](#)
[Don't Trust Blindly](#)
[Investigate Assumptions and Dependencies](#)
[Verify Your Assumptions](#)
[Learn About the Context](#)

17. Experiment

[Conduct Exploratory Code Modifications](#)

[Refine and Test Hypotheses](#)

[Restore Your Code to Its Original State](#)

18. Keep a Debugging Log

[Track Your Progress](#)

[Avoid Distractions](#)

[Use Documentation Tools](#)

[Keep an Archive](#)

[Use Chats as a Log](#)

19. Build Your Own Tools

[Implementation](#)

[Recommended Features for Debugging Tools](#)

[Promotion and Documentation](#)

[Release Considerations](#)

20. Build a Minimal Repro

[Choose a Minimization Method](#)

[Minimize the Input](#)

[Share the Example](#)

[Prevent Regressions](#)

21. Debug with printf

[Avoid printf Pitfalls](#)

[Know When to Use printf](#)

[Implement printf](#)

[Leverage Advanced Logging](#)

[Clean Up](#)

22. Use Off-the-Shelf Tools

[Consider Availability](#)

[Know When to Reach for a Specialized Tool](#)

23. Talk It Through

[Define Your Goal](#)

[Choose Who You Talk To](#)

[Plan the Logistics](#)

[Decide What to Cover](#)

[Summary](#)

PART IV: FIX

[What Makes a Great Bug Fix](#)

[Make It Revertible](#)

[Address the Root Cause](#)

[Verify](#)

24. Just Fix It

[Remove the Root Cause](#)

[Change the Original Sources](#)

[Test for Side Effects](#)

25. Share What You've Learned

[Document the Fix](#)

[Find What Produced the Root Cause](#)
[Prevent the Bug from Happening Again](#)

26. Search for Related Problems

[Understand Patterns and Dependencies](#)
[Make Bulk Changes with Caution](#)

27. Fix Early

[Assign Reasonable Priorities](#)
[Leverage Context](#)
[Prepare for Attrition](#)
[Avoid Dependent Behavior](#)
[Take Ownership](#)

28. Deploy a Quick Fix

[Undo the Offending Change](#)
[Implement a Workaround](#)
[Communicate Workarounds](#)
[Don't Stop at the Workaround](#)

29. Start Over

[Rewrite the Offending Functionality](#)
[Test the New Implementation](#)
[Compare the Results](#)
[Stub the Implementation](#)

30. Find a Creative Solution

[Declare It a Feature](#)
[Remove the Feature](#)
[Remember to Communicate](#)
[Just Let It Be](#)

[Summary](#)

CONCLUSION

INDEX

THE BOOK OF DEBUGGING

A Systematic Workflow for Finding and Fixing Bugs

by Johannes Kuhlmann



THE BOOK OF DEBUGGING. Copyright © 2027 by Johannes Kuhlmann.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

First printing

30 29 28 27 26 1 2 3 4 5

ISBN-13: 978-1-7185-0406-6 (print)

ISBN-13: 978-1-7185-0407-3 (ebook)



Published by No Starch Press®, Inc.
245 8th Street, San Francisco, CA 94103
phone: +1.415.863.9900
www.nostarch.com; info@nostarch.com

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Sabrina Plomitallo-González
Developmental Editors: Eva Morrow, Abigail Schott-Rosenfield, and Rebecca Senninger
Cover Illustrator: Rob Fiore
Interior Design: Octopod Studios
Technical Reviewer: Steve Rabin
Copyeditor: Isabel Kunkle
Proofreaders: Cindy Snyder and Michele Kessel

Library of Congress Control Number: 2026024610

For customer service inquiries, please contact info@nostarch.com. For information on distribution, bulk sales, corporate sales, or translations: sales@nostarch.com. For permission to translate this work: rights@nostarch.com. To report counterfeit copies or piracy: counterfeit@nostarch.com. The authorized representative in the EU for product safety and compliance is EU Compliance Partner, Pärnu mnt. 139b-14, 11317 Tallinn, Estonia, hello@eucompliancepartner.com, +3375690241.

No Starch Press and the No Starch Press iron logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc.

shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

About the Author

Johannes Kuhlmann is the studio technical director at Fishlabs and has more than 15 years of experience as a programmer and technical lead. Kuhlmann has worked on a wide range of video games, including *Chorus*, *Goat Simulator 3*, *Valheim*, and titles from the *Saints Row* and *LittleBigPlanet* franchises. He holds a Diplom in computer science from the University of Hamburg. Having learned debugging by trial and error, he has prevented innumerable bugs from reaching users.

Feel free to share your own debugging strategies with him via email at debugging@johanneskuhlmann.de.

About the Technical Reviewer

Steve Rabin is a principal software engineer at Electronic Arts, where he specializes in optimizing games and developing performance tools as part of the Performance Technology Group. He was previously a principal software engineer at Nintendo Technology Development, where he oversaw UX design for developer tools on the Nintendo Switch and Nintendo Switch 2, as well as architecting tools such as the Nintendo CPU Profiler for the Wii, Wii U, 3DS, and Nintendo Switch.

Rabin is an inventor with 16 issued patents covering data visualization, profiling, augmented reality, and game controllers. He is an advisor and founder of the GDC Game AI Summit, the editor of the Game AI Pro book series, and a computer science instructor at DigiPen Institute of Technology. He earned a BS in computer engineering and an MS in computer science, both from the University of Washington.

ACKNOWLEDGMENTS

Writing this book has been a lifelong goal come true. It was only possible to achieve with the direct and indirect help of many people, and I will just name a few here.

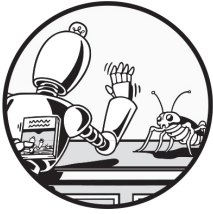
First, I must thank Insa, who helped me sort out many of my thoughts, strategized with me on how to get the book done, and supported me spending all this time on it. I'd also like to thank my children, Eik, Mats, and Jarno, who were always encouraging and who I hope one day will read this book and learn a thing or two from it.

My buddy Chip has been involved with the book in some way or another since the beginning. I've discussed many of the concepts with him, and he was always a great source of feedback.

My editors, Abigail, Eva, and Rebecca, have been instrumental in getting the book into shape. They made it readable and suggested many, many great improvements. Steve, my technical reviewer, not only provided a lot of fruitful feedback but also helped me with further discussions about details and terms.

Lastly, I must thank all the people I've worked with over the years. They gave me a lot of input, were great sounding boards, and enabled me to test my ideas in practice.

INTRODUCTION



When I was starting out in software development, I was working on an educational game that had a built-in dictionary. To provide a more responsive feel, I multithreaded the search so that users could keep typing while the results loaded. On very rare occasions, the program crashed when users closed the dictionary after searching for a word.

I was young and confident in my skills, so I got out my debugger and tried to reproduce the problem. I also tried finding a reliable reproduction with concrete steps and went through the code to verify that I hadn't made any mistakes. Nothing.

I ultimately automated the reproduction. I changed the game to go into the dictionary module, search for some words, leave the dictionary, and repeat. I was sure if I let the automated test run over the weekend, I'd find a nice, clean call stack in the debugger when I returned on Monday, telling me exactly where the problem was.

When I came in on Monday? Nothing. The game was still happily trying to reproduce the bug. No crash.

I went through the code again and added some mutexes for safety (even though I was sure I wouldn't need them). The bug became even rarer, but it still happened occasionally. We ended up shipping the program in this state.

I didn't have a workflow. I had a debugger, some intuition, and a willingness to try things, but I didn't have a systematic way to find a bug

that didn't want to be found. This book is the workflow I wish I'd had then: a sequence of steps and a set of strategies for working through bugs that resist the obvious approaches.

We can't avoid bugs entirely. Programming languages, frameworks, libraries, tools, operating systems, hardware, and requirements are too complex for any single person to understand completely. Developers, educators, and researchers are working to create fail-safe languages and tools to prevent bugs in the first place, but that's not what this book is about. This book is about what to do when, despite everyone's best efforts, the bugs are still there.

Fixing difficult bugs is like a huge puzzle: If you keep finding more and more pieces that fit together, you'll have the complete picture at some point. Even when there might be too many pieces and too little time to get all of them sorted and identified, you must remind yourself that it's fundamentally possible.

To be honest, I enjoy debugging quite a bit. It's such a great feeling when you achieve the seemingly impossible and fix that super-hard bug that was keeping people from using your program effectively—or at all!

What Is a Bug?

Talking about bugs and debugging raises the obvious question of what a bug actually is. We could go to some lengths to find the one correct and official definition that is true for all software, perhaps by diving into the relevant ISO standard (ISO/IEC/IEEE 24765:2017, a reference of common vocabulary in systems and software engineering). However, this doesn't necessarily cover every kind of bug. For example, is a feature not feeling *entirely right* something that would fit the formal definition of a fault in a software product?

Bugs can manifest in many ways: crashes, slowness, broken features, visual glitches, audio problems, data leaks, and more. Even if a program has perfect technical implementation, bugs can reduce its fidelity and usefulness. For this book, let's keep it simple. We just want our software to be great, so we'll call anything that's not behaving as intended a bug.

Why We Should Fix Bugs

There are many reasons to fix bugs:

Quality matters While annoying little bugs might not directly affect a program's usability, they can make it look cheap or low effort. Bugs in a program don't instill trust in its abilities. Even if there are only minor issues, like text kerning being off or rare, random crashes, your users might quickly look for an alternative. If your program doesn't deliver on its core functionality, that's even worse. You must provide a quality product right out of the gate to get noticed and make a difference in a crowded market.

Bugs slow down development A bug in an internal tool that developers use frequently can severely hinder the development process even if users never see it. One that slows down testing will hurt the performance of anyone who is testing the program or executing it as part of their development workflow.

Bugs can spread Even though computer bugs don't propagate by themselves, they can still spread through reuse or copy-and-paste of the offending code or asset. This even happens across projects, as programmers reference existing code or reuse small pieces from other software. Such code can provide simple functionality like getting the current time, generating a random number, or sanitizing user input; or it can be much more project-specific. Fixing bugs earlier can prevent this from happening.

Aging bugs increase complexity The more recently a bug was added, the easier it is to fix. The programmer who wrote the code is most likely still around and remembers the change, so it's a lot easier for them to pick it up again and fix it. Other programs might start depending on the buggy code, making it harder to fix without breaking dependencies.

Bugs can expose private data When programs deal with personal data, developers must follow established best practices to protect it, even when breaches don't directly affect a program's quality. Leaking personal data means losing users' trust, maybe permanently, and might also mean fines in some parts of the world.

High usage magnifies bugs Let's say a bug takes a single tester one day and 100 tries in total to reproduce; that is, it has a 1 percent reproduction rate. This incidence seems very low. However, if your program turns out to be successful and you have 100,000 concurrent users, then 1,000 will be exposed to this particular bug every day.

Debugging Is Inevitable

Debugging is something that you have to do constantly while developing software. It is part of the development process while you iterate on the functionality, it comes up as testers or users report problems, and it takes up a lot of programmers' time.

Debugging has been a substantial part of programmers' work since the very beginning. In *Memoirs of a Computer Pioneer* (MIT Press, 1985), Maurice Wilkes writes:

By June 1949 people had begun to realize that it was not so easy to get a program right as had at one time appeared. [...] The realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.

In *The Elements of Programming Style* (McGraw Hill, 1978), Brian W. Kernighan and P.J. Plauger point out the difficulty of debugging:

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

How much time you spend debugging depends heavily on your project, your role, and what stage of the development process you're in. For a little-used program where you're writing new features in a relatively safe language, you can likely get by without much debugging at all. If you're modifying a popular program that wasn't especially bug-free to begin with, or porting one to a new platform, you can easily burn a lot of time fixing both new and existing bugs. If you're in a senior leadership or mentor position, you're likely involved in fixing bugs that more junior developers find hard to address, splitting time between other non-programming tasks and helping the team sort out bugs.

In the beginning stages of a project, you'll focus on writing new code, and while that involves making sure it works correctly, you're probably

spending most of your time implementing features. Toward the end of the project, while you're polishing the program, you'll use almost all your time fixing the most critical bugs.

Debugging Is Your Job

Let's say somehow you happen upon a bug. You may have found it yourself, your quality assurance (QA) team may have reported it, your users may have sounded the alarm, or the bug may have taken one of a thousand other possible routes to you. Regardless, it's hopefully fair to assume that you have some kind of description of the bug (even if it's just in your head) and a vague idea of the circumstances it appears in. What should you do next?

First, accept that you're now responsible for fixing this bug. It's almost too easy to blame problems on something or someone else. For example, you might want to attribute the bug to user error, the compiler, or some other team member's code, but it's more likely that code written for a specific project is faulty. A third-party system that others have tested and used is less likely to be broken. That's why you must assume that a bug is your (or your team's) fault and step up to fix it.

Another common mistake is to push back on dealing with a bug because, for example, you don't feel like fixing it, you don't see enough detail in the description, you aren't comfortable performing the inspection, or you even assume that, surely, this functionality simply can't be broken. However, if a bug was sighted, it's probably real. You may not have all the information you wish for, but you have to live with that to keep improving your program. In practice, go even further. As a routine, keep an eye on the bug tracker and request assignments to bugs that are related to your work or that you feel best suited for investigating. Take ownership of what you've implemented and what your team has built.

Who This Book Is For

This book is meant to fill the gap in existing educational material about debugging for software developers. Even though debugging takes up much of a developer's time and is imperative to shipping a successful program, minimal effort goes into training, coaching, and educating professional or aspiring developers.

While the underlying assumption is that you're a programmer, it's not a hard requirement, as quite a few strategies will be useful for non-programmers. To follow along with this book, however, it's helpful if you have an idea of how programming works.

Developers with a background in other disciplines will find helpful techniques that are applicable even if you work in a different field like art production, interface design, or QA. Debugging involves systematically solving a complex problem, which is frequently possible without touching the source code. You have to collect information on the issue and analyze it in the context of the bug to identify the underlying *root cause*, or the factor that, if eliminated, would prevent the bug from occurring.

When we think about debugging, we might first imagine a traditional programming bug that needs to be solved by a programmer modifying the source code. However, as frameworks and tools become increasingly capable, they put more power—and greater responsibility—in the hands of artists, designers, and people from other disciplines. If you have more power to do things by yourself, you're also responsible for doing them right.

Prerequisites

You should be comfortable using a version control system (Git, Perforce, or Subversion) and a project management tool with bug tracking built in (there are many options, including Jira, Asana, Redmine, and ClickUp). The version control system tracks your project's history so you can see what changed and when. The bug tracker holds the description, reproduction steps, status, and other data you'll need to resolve each bug. If you're not already using these, get them. This book will help you a lot more if you do.

How to Use This Book

You can read this book straight through or use it as a reference whenever you're stuck. Experiment and keep the advice that resonates with you. The strategies presented here are generally platform- and language-agnostic. In some cases, I'll go into more detail about standard technologies used in particular programs, but that doesn't mean the method in question won't work in other situations. Even the more technical strategies don't go into very low-level details. For example, analyzing assembly code is rarely necessary. It's usually more beneficial to reason about a bug on a higher level and figure out where the logic is flawed.

This book is not about preventing bugs or improving your programs. It addresses situations where a bug is present and needs to be found and fixed, using a structured workflow.

This book is organized as follows:

The Workflow Covers the general workflow and discrete steps you can use as a guiding framework to approach new bugs. Additionally, this chapter provides a rundown of common root causes or categories of bugs to keep in mind while you're progressing through the workflow.

Part I: Reproduce Discusses how to reproduce a bug and what information to include in a report. This part also describes ways to make reproduction easier and faster than simply using your program.

Part II: Probe Shows how to find a suitable entry point for a detailed investigation. Methods include using your experience to identify shortcuts, searching the internet, discussing the problem with other people, and incorporating a debugger into your efforts.

Part III: Examine Covers how to examine your program and its sources. The debugger comes into play here as well, as do strategies like creating minimal versions of the program, using external tools, and analyzing the code. It's important that you question all of your assumptions during this stage.

Part IV: Fix Explains how to fix the bug that you've investigated, and whether you need to do so urgently or can take your time to produce sustainable results. If a clean solution isn't possible, you can

implement a hacky workaround, reimplement the functionality from scratch, disable the whole thing, or just live with the bug.

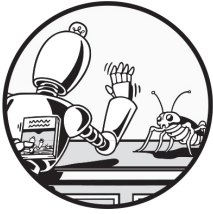
Manage Expectations

Before we venture deeper into the realm of debugging, a warning is in order. Working on bugs typically involves a lot of stress and pressure. Aside from the rare occasions when you encounter a trivial bug or make a lucky guess, the repair process also takes quite a bit of time and persistence. Users may have unrealistic expectations of how easily and quickly you can fix an issue, which adds to the pressure, as estimating resolution time is notoriously difficult.

Manage your coworkers' and users' expectations accordingly: It's always better to under-promise and over-deliver. You'll need to manage your own expectations as well. Working through the bug systematically, using an established workflow and taking your time, will help keep you from overlooking anything. Make sure you have an environment where you can focus, since debugging has a very high cognitive load. If you feel stuck, it may be time to call it a day and continue after a good night's sleep.

Fixing bugs is difficult, but there is hope! The computers we use are deterministic, so bugs can't be true random events. Any bug in your program must exist for a valid reason. The reason may be complicated, and the bug's symptom(s) may be related to the root cause only via several steps of indirection, but there must be a connection. You just have to trace it to find the root cause, and this book should simplify that process.

THE WORKFLOW



You've probably had this happen: a bug shows up, and you start poking around, checking a likely spot, trying a change, checking another spot, trying another change, and an hour later you're no closer and you're not even sure what you've ruled out. That's what debugging without a plan looks like. The problem is open-ended by nature; nothing about a bug tells you where to start or when you're done. A workflow gives you that structure, so your effort goes somewhere instead of scattering.

Having a workflow helps in three ways. It ensures you gather every essential detail, it splits the work into distinct steps so that you can focus on each one (and often finish faster), and it gives you confidence that the bug is genuinely fixed.

Once you've accepted responsibility, it's time to investigate and fix the bug. [Figure 1 shows](#) the general workflow you should follow, and the subsequent sections describe these steps in detail.

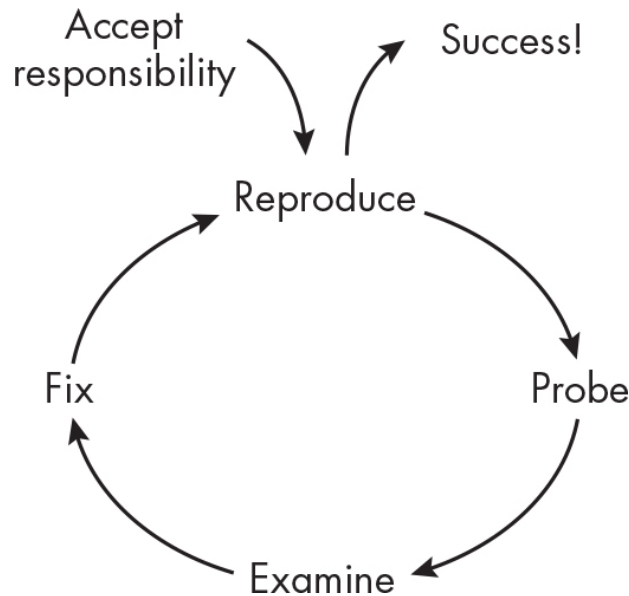


Figure 1: The four distinct steps in an iterative debugging workflow

You will go through one or more iterations of this workflow until you've found a solution for the bug and can't reproduce it anymore. While the numbers suggest you must perform the steps in a particular order, that's not always the case. It's true, however, that you always have to start with reproducing step. Otherwise, you can't effectively investigate anything.

Step 1: Reproduce

The very first part of fixing a bug is trying to reproduce it, often referred to as a "repro" or "repro step." This first step is one you can't really skip. If you can reproduce the bug, that's great, and you can continue to the next step. If you can't reproduce it, you must keep trying, varying your actions until you're successful. Otherwise, fixing the bug correctly will be more luck than anything else. A bug that has been reported is most likely real. If it's difficult to reproduce, chances are you might be overlooking an important factor. If you can't reproduce it, though, you have to send it back to the QA team or original reporter, who will verify that it doesn't exist anymore (or never did).

There certainly are bugs that you can't reproduce in investigation and debugging environments. Such cases require you to separate reproduction in a test environment from investigation and fixing in the development

environment, but you still need to be able to reproduce the bug so you can confirm that your fix works.

Being able to reproduce a bug is half the battle, but once you know what triggers it, you can use many different strategies to find out what's going wrong.

Step 2: Probe

After reproducing the bug successfully, the next task is to probe the program for a related entry point, which can be anything connecting the bug's symptom(s) to the program's sources. Multiple layers of indirection can obscure this connection, but you have to identify it, because those sources are the only places you can fix the bug.

Finding the entry point is like finding the end of a tangled thread, probing electronics for faulty components, or surveying a piece of land for natural resources. It's all about covering a large amount of ground in a relatively short time to find a spot where you can start digging deeper.

While most bugs stem from the source code in practice, they may also exist in other building parts of your program, such as an incomplete or incorrectly installed asset. Bugs could also come from faulty configuration or documentation. That's why it's essential to keep an open mind and look for an entry point in all the program's sources.

Ideally, the guidelines and tools you use to build your program would make it difficult to introduce bugs anywhere, but it's usually impossible to prevent all mistakes. Apparent mistakes may also be intentional, like a computer flagging a logo as erroneous when it's simply a variation for a different use.

While probing for an entry point is closely related to the next step, approaching it separately is helpful. Probing involves looking at the bigger picture to narrow your focus. Once you know a source related to your bug, you can move on to the examination.

Step 3: Examine

After finding a potential entry point, you must examine it closely and follow any leads to find your bug's root cause. The problem is that the

reason for the bug's symptoms—whatever is really to blame for the unwanted behavior—can be hidden behind several layers too.

Because the root cause can be in any of your program's sources, you may have to go through several connections that take you through different areas of interest, which may lie in different kinds of building parts. For example, you may have identified a source code function connected to the bug symptoms during the probing step. You could change it not to show the symptoms, but digging deeper could reveal the real cause: data from a broken asset.

If you started off knowing the complete context of the bug and all the inputs that contribute to it, the solution would be obvious. Unfortunately, that's rarely the case. Instead, you have to work with limited information and extend it wherever there are gaps relevant to the bug at hand. That's precisely what this step is about.

In contrast to probing for an entry point, an examination is very detail focused. You need to be able to analyze the smallest parts individually and see how they come together to form a complete system. You must follow connections between those separate parts to understand the bug as a whole, which is why the probing and examining steps are often done in short iterations. It's also common for bugs to span multiple components of a program, so you may have to chase one through the assets, the code, the configuration, and so on.

IN PRACTICE: FOLLOW REFERENCES TO GET TO THE ROOT CAUSE

One bug I dealt with popped up on an internal graphical user interface (GUI) application. When a user pressed a particular button, the program took quite a while to process before the bug occurred. I started debugging by looking at the asset describing the window that contained the button and found that it referenced a certain code function. Navigating to that function revealed that the program used a particular file to establish the state it was in the last time a user closed it; opening that file showed me why a specific step in the program wasn't being executed as expected. I had to return to the code to fix another function responsible for writing the state to the file.

I debugged all these different areas with different tools. The problem had several layers; only following through with all of them told the whole story and allowed me to identify the root cause.

Navigating your project should take you closer to the root cause of the bug if everything goes right. (If you end up in an unrelated spot, you must backtrack to the correct path and take another direction.) Connections are mostly function calls and data dependencies.

How do you know you've arrived at the root cause, though? Ideally, you have acquired a full understanding of the bug so that you know exactly how it came to be, how it triggers its symptoms, and, therefore, what the root cause is. Unfortunately, investing enough time to get to this point is often impossible. In such cases, the root cause is the spot where you can make changes that fix the bug and there are no obvious underlying areas that could be responsible.

Step 4: Fix

With the root cause nailed down, all that you have left to do is fix the bug. Sometimes, this is straightforward, such as correcting a typo or making a small, localized change. There might still be adverse side effects, but those should be easy to understand and trace.

A more extensive fix requires changes in several places. It can take quite a bit of time because the changes may affect many different areas, which you have to understand first. For example, a whole system may have even been implemented with the wrong central underlying assumption.

Perhaps the system needs to be safe for use in a multithreaded context, which it wasn't designed for. This would prompt a complete refactor of the system and possibly even a change to its interface. The modifications would require a lot of work and would be relatively widespread. Likewise, if all text has to be encoded in UTF-8 instead of ASCII, the change will affect all the areas in the code that need to deal with text. You'd need to deal with it in many different locations, without any means of isolating the impact.

Because it can be challenging to resolve a bug perfectly (relative to the requirements for the program) and because other people might be waiting on the result, you should be flexible regarding the quality of your fix. You can invest the work to understand the bug's entire context perfectly, but that will probably take quite a bit of time. Alternatively, you can implement a hacky workaround that eliminates the bug's symptoms and allows anyone who is blocked to continue their work while you continue to search for a

more sustainable fix. The level of urgency will determine the right approach. Making this choice is a tricky balancing act, but over time, you build the experience to make a good call.

In any case, once you've implemented a fix, you must verify it by returning to the workflow's first step and trying to reproduce the bug again. If you can't reproduce it, then awesome! You're done. Otherwise, you'll continue this loop until you can no longer reproduce the bug.

You may also reach a point where you're simply unable to fix a bug because you don't understand its context or lack the skills to address it without months or years of training. In such situations, it's best to identify the person who would ideally be working on this bug and pass it on so they can fix it.

Handle Bugs as a Team

So far, we've focused on what the person investigating an already reported bug should do, but how should the whole process around fixing bugs work?

A dedicated QA team should focus on testing a program as it's being built and identifying potential issues as early as possible. While this naturally includes fixing conventional bugs, it must also address usability, accessibility, and specification fulfillment. Such bigger-picture bugs are less critical at the beginning of development than they are toward the end, when the program's overall quality takes center stage.

Developers should establish processes and guidelines for working with the QA team. Most importantly, all bug reports should go through a bug reporting tool. Most communication around bugs should happen via this method so that the communication and the bug itself are documented in the same place.

When a QA tester discovers a bug, they need to find out as much as possible about it and write it all down. A great tester always records exactly what they were doing so that they can retrace their steps and, in many cases, reliably reproduce the bug. The tester should try reducing the steps to the bare minimum so they report only those relevant to a specific bug. See "What Makes a Great Bug Report?" on [page 11](#) about what other data testers should include. Once a bug has been recorded, it must be assigned to someone. Depending on the organization, it might go first to a project

manager for prioritization, to a lead developer for input and theories before assignment, or directly to the relevant developer if the team is small enough for the tester to know who that is. Distribution can happen immediately when the bug is reported or can be scheduled through the development process. For example, one common approach is to assign bug fixes during team planning meetings. An important rule is that a bug must be assigned to only one person. Otherwise, either the bug gets buried because nobody feels responsible or multiple people duplicate each other's work. The one assigned developer follows the debugging workflow and (hopefully) fixes the bug. When the developer is done, they mark the bug resolved and assign it to the original reporter. It's imperative that the person responsible for debugging never single-handedly closes the bug. It needs to go back to a neutral party to verify the fix.

A few different outcomes can result from the process. The developer may mark the bug as "fixed" and have a tester verify it. They may mark it as "unable to reproduce," meaning the QA team has to determine whether they can reproduce it and update the reproduction steps accordingly. Finally, the developer may mark the bug as "won't fix," which means they, together with all relevant stakeholders, have decided not to bother addressing it. In this case, no further testing is required.

If the same bug or a very similar issue comes up later, this bug report should be reopened so that someone investigating it has the relevant history and can let previous investigations and possible fixes influence their new debugging adventure.

Common Root Causes

After looking at many bugs and investigating them down to their root causes, you'll start to notice they often fall into categories. Depending on the type of program, the programming language, or the environment your program operates in, some causes may be very rare or irrelevant. The following non-exhaustive list of common root causes should serve as a starting point to build your expertise:

Bad Data

Some of the data your program deals with can cause it to misbehave or deliver incorrect results. This can include information that doesn't make sense in context, like the wrong coordinates for a button that position it off-screen. Bad data can also have been corrupted somewhere, such as the asset on your hard disk, your program, or another RAM process. In such a case, it probably contains more or less random elements.

Bad Pointers

A pointer could hold the wrong memory address. Bad pointers occur when:

- A pointer is never assigned an address or is assigned the wrong one.
- A pointer isn't updated even though the data it points to is moved.
- A pointer is still used even though the data behind it has been removed.
- A pointer's arithmetic goes wrong.

There is also the infamous *null pointer* error, which means the pointer is set to the 0 address to signify that it's invalid. Unfortunately, the possibility of encountering this issue means you always have to check whether a pointer is null before you access it.

While a bad pointer is a special case of bad data, it's essential to call it out explicitly because it's a class of bugs that is extremely widespread in languages with unsafe memory management. For example, bad pointers are among the most common root causes of bugs in C and C++.

Bad Logic

A feature or system in your program might behave incorrectly without specifically crashing. For example, a progress bar might show a process is completed before it's actually done, a button might open the wrong dialog, or a calculator might provide incorrect results.

Edge Cases

Edge cases occur at an extreme (maximum or minimum) operating parameter or boundary condition of a system where typical assumptions no longer hold. They can be problematic because it's common to think only about the core logic when implementing new functionality. One typical example is an off-by-one error that occurs when iterating over all elements

in a collection. Leaving out the first or last element can easily happen, and it's rarely evident because the result is almost correct. Other common edge cases include chronological problems caused by switching to daylight savings time or changing time zones and issues that arise when a program is killed without a clean shutdown.

Floating-Point Inaccuracies

Computers usually adopt floating-point numbers to represent real ones. However, when it comes to very small or very large numbers and when accuracy becomes important, problems can show up. These include jittering positions or movements in your graphical output or the inability to accurately position a user interface (UI) element.

Two floating-point formats are commonly used: `float` and `double`. A `float` spends 32 bits on its representation, while `double` uses a whole 64 bits. Swapping out a `float` for a `double` can be a quick, straightforward fix that gives you some breathing room and might fix the bug altogether.

See “Follow the Data” on [page 122](#) for more potential issues with floating-point numbers.

Stale Code or Assets

Stale code or assets can affect both published builds of a program and implementations under development. A situation in which a program is operating with stale code or data could occur because the program hasn't incorporated your changes to the code (leaving you building one configuration but running another), because the program isn't properly processing the assets, or because you're changing the wrong file. Other causes could be the deployment step not putting the build output in the right location, a source file or target file being marked read-only, or a file being in use. The version control branch with your changes could also be the wrong one or not have been merged yet. The question is whether the modifications to your program's sources actually make a difference. Sometimes that's not the case, and you don't notice immediately. You may keep experimenting (see “Experiment” on [page 134](#)) or try to implement a fix without success. It's only later that you realize the program didn't pick up your changes and you must start from the beginning. To avoid this situation, you should make very obvious changes and build the program as

a test before continuing (see “Instrument the Program to Report Problems” on [page 62](#)).

Relatedly, make sure you push updates out regularly. Users working with old releases may end up exposed to bugs you’ve already fixed.

Wrong Timing or Order

Dependencies (wherein some steps depend on the results of others) often aren’t obvious and can be obfuscated by layers of indirection, making it nearly impossible for you to analyze them. The order may be wrong, such as when the step that should be the second one comes before the first.

Alternatively, the timing might cause the program to miss important events, not let the user react, or otherwise lead to problems. Two steps may also run simultaneously, leading to the data not being available in time or insufficient time between the steps for the results to be ready or usable.

Bugs like these become even more complicated to detect and examine when the order or timing isn’t always the same, and the bug, therefore, becomes random (see “Remove Randomness” on [page 69](#)). While such bugs occur frequently enough in single-threaded programs, they’re more likely and severe in multithreaded software.

Unclear or Bad Requirements

With unclear or bad requirements, there isn’t anything wrong with the program or its sources. The code and assets can fulfill all the initial requirements, and they may even have been appropriately tested and gone through several iterations.

The problem is that either the requirements are simply wrong or the people behind them stated them ambiguously. While the people implementing the functionality followed the specification to the best of their abilities, they couldn’t deliver what the user actually wanted.

Addressing this issue means clarifying the requirements, which may involve more people than those working on the program directly.

User Experience Issues

Even when everything is technically correct, from the design and requirements down to the implementation, users may not be pleased with the feel of the program or the way a feature or system fits in. Likewise,

game developers may remove perfectly implemented features if they're just not much fun.

User experience (UX) covers how easily, satisfyingly, and effectively people can operate programs. QA testers will report UX problems as bugs, and you'll need to fix them the same way.

Summary

This chapter introduced the debugging workflow: the four steps you'll take to get from a bug report to a working solution: reproduce, probe, examine, and fix. It also walked through some of the most common root causes behind bugs, so you'll recognize them faster when they turn up. The rest of the book takes each step in turn, with specific strategies you can reach for at every stage. Whenever you're stuck, you can come back here to see where you are in the process.

Part I

REPRODUCE



The first step in debugging is to reproduce the bug. Reproducing it confirms the bug is real and lets you gather information firsthand instead of relying on someone else's account. The more you can see for yourself, the easier the root cause is to find.

Reproducing a bug starts with a good bug report, so this part begins there: what makes a report useful and what to ask your reporters for. From there, it moves to the bug itself: how to make it happen reliably, how to make it happen faster, and how to get the computer to reproduce it for you, so you don't have to do it by hand. By the end, you should have a bug you can trigger on demand, which is the foundation for everything that follows.

What Makes a Great Bug Report?

The most effective bug reports lead you directly to the source of the problem. In the best-case scenario, the reporter's description will guide you through running the program in a debugger, reproducing the bug by following the given steps, identifying the issue in the code, and fixing it. Though most cases won't be this simple, the better the bug report is, the easier the debugging process will be.

To ensure your bug reports are as helpful as possible, talk to the reporters about what data they should provide. Getting end users to collect diagnostic information can be a challenge because they often lack expertise in doing so and have little incentive to invest a lot of time in the task. That's why you should streamline the process and ask only for essential facts.

Larger companies normally employ experienced, professional QA teams rather than relying on end users. While such teams often have standard checklists of information to include, it's important to tell them any specifics that would be most helpful to you; otherwise, those details might not make it into their reports.

Start with the following categories of data:

Symptoms An objective description of anything the reporter observes when the bug happens, without including theories about the potential reasons those events occur. Such hypotheses could bias the investigator and prevent them from finding the bug's actual cause.

Expected behavior A detailed description of how the reporter expected the program to function. This helps identify cases where someone mistakes intended but surprising functionality for a bug.

Reproduction steps A concise, enumerated list of every step you must execute to reproduce the described symptoms, starting when you launch the program and ending with the bug and its effects.

Reproduction frequency A rough indication of how often an issue happens. The reporter should reproduce the bug several times, noting the number of attempts and how many were successful (they might express this as a percentage). Alternatively, they can categorize reproduction frequency as always, frequently, rarely, or only once.

Severity A subjective assessment of how disruptive the bug's symptoms were. This helps developers and managers decide how to prioritize it.

Platform Descriptions of the platform (including specific details if the tester isn't working with a fixed hardware configuration) and any other relevant information about the situation. Since you can't expect the reporter to test all environments in which the program might potentially run, the listed reproduction steps are usually valid only for

the environment and circumstances in which they originally observed the bug.

Build details All the information that can help identify the specific build used for testing. This includes the version number (if there is one) and, ideally, the exact commit the build was based on. It should also cover how the build was created: the configuration that was used (debug, release, and so on) and whether any custom changes were applied.

Previously attempted workarounds or solutions A list of steps that the reporter has already taken to fix or bypass the issue. Knowing what's been attempted can rule out certain causes or point toward others.

History of previous investigations A reference to the older bug report, if the same or a very similar bug has been filed before. The reporter might also choose to reopen the old report and add any newly observed information, which is better than starting a new report if the old and new bugs are identical. The reporter should search the list of existing bugs before starting a new ticket to see if any could be relevant.

Additional data Any other available details the reporter comes across and deems relevant, even if the report doesn't directly request them. Recording the reproduction on video or capturing a screenshot can clarify descriptions and add detail to reproduction steps. Likewise, if the reporter has access to a crash dump (see “Collect Crash Dumps” on [page 91](#)), an input file, or another saved state that helps reproduce the bug, they should attach that to the bug report as well.

Contact information The reporter's name and a way to reach them. If you need more information while investigating the bug, you must be able to contact the reporter and ask questions.

The following box shows an example of a bug report that features all the important information.

BUG #235

Description: Settings issue.

Observed behavior: The settings don't contain an offline/online option.

Expected behavior: The user should be able to switch between online and offline modes in the settings.

Additional comments: Version 259390 from last week still had the option. I tried the program on different platforms, but the behavior was the same on all of them.

Steps to reproduce:

1. Launch the program.
2. Make sure you're signed in with an account that has online privileges.
3. Open the settings menu.
4. Observe that there is no offline/online setting.

Status: Assigned

Assigned to: Johannes

Reported by: Steve

Reproducibility: Always (5/5)

Severity: Major

Found in version: 267023

Platform: All

Priority: High

Attachments: settings_screenshot.jpg

History:

2026-10-12 4:12pm Steve created the bug report.

2026-10-12 5:30pm Abigail assigned the bug to Johannes.

A report's comprehensiveness depends on how much time the reporter can invest in creating it. While more information improves the chances of a fix, it also heightens the barrier of reporting the bug in the first place. You must find a balance between these two factors. Separate all the data you absolutely need as mandatory and mark the rest optional.

What you should consider mandatory depends on the project and the bug itself. For example, you will need to know what graphics card the tester was using if a graphical real-time program performs badly, but probably not if a command-line application crashes on the CPU. Because you can't assume you'll be the last person looking into this particular bug, add anything you find out to the report. Having information from previous investigations will make your life or your successor's a lot easier. Be sure to log the bug report and all its data in your bug tracking or project management software.

Keep in mind that bugs can manifest in many different ways. The implementation, logic, or original requirements may be wrong, or a user might expect the program to behave differently even if everything works as intended.

While the original bug report hopefully tells you which manifestation you're dealing with, you can add more detail when reproducing the issue yourself, especially if you have better insights than the reporter into how the program works internally. The repro step also confirms that another team member hasn't fixed the bug already and that it actually existed in the first place, rather than arising from misunderstanding or user error.

These questions can serve as starting points:

- Does the bug really exist?
- Is the bug's behavior intentional (meaning the bug isn't really a bug)?
- How exactly does the bug manifest?
- What steps or conditions are required to reproduce this bug?
- How frequently does the bug occur?

IN PRACTICE: INTENTIONAL BUGS

Sometimes, it can be hard to tell whether an apparent bug is actually intended behavior. I once worked on a game that simulated graphical corruption and another that mimicked Windows's blue screen of death.

If your program shows or simulates behavior that might be mistaken for a bug, you need to communicate that to outside testers. This is especially critical when platform holders or storefronts are testing your program and a failure might lead to increased costs or a delayed release. If you suspect a program you're testing imitates bug symptoms on purpose, check with your team and go over any existing documentation. If you can't make the bug happen, you'll only be guessing at its causes and potential solutions, and you'll never know whether you really fixed it.

1. Reproduce the Bug Yourself

How? Emulate the reporter and follow the reproduction steps as closely as possible. Then introduce variations and explore how they affect the bug.

Why? You get to experience the bug just like the reporter encountered it and a user would observe it, which allows you to learn more about the bug and its environment. By trying variations, you can determine what reproduction steps and details are essential.

Simply using the program is the most straightforward way to start reproducing the bug you're about to fix. It requires no special knowledge, skill, tooling, or setup, so you can start immediately.

Re-create the User Experience

Using the program allows you to experience the bug the way the reporter originally encountered it, closely observing the circumstances that triggered it and what prerequisites have to be in place. If you haven't (recently or ever) used the feature or component of the program where the bug occurs, you'll also gain potentially helpful background information about its context.

IN PRACTICE: OBSERVE THE BUG

A project I worked on received a bug report saying a network connection had been interrupted when an input device was disconnected. If I had approached this bug without any further background information, I might have ended up debugging my network code to find out how input devices could influence it. By reproducing the issue myself, I quickly came to a different conclusion: When the input device was disconnected, it triggered a modal dialog, preventing the network code from being executed so that the connection timed out and was interrupted. That would have been hard to see when debugging the network code, but using the program and not being able to interact with anything due to the modal dialog made the problem painfully obvious.

As you work, fill in observations that are missing from the report. How exactly does the bug manifest? What does it look like? Others looking into the same bug in the future will also benefit from this information. As you develop a better understanding of the bug, reevaluate its impact. If the bug is minor or happens only in very particular situations, fixing it might take a

lower priority than other tasks. However, discuss that prioritization with the original reporter in case you overlooked an important factor and misjudged the bug’s severity.

Emulate the Reporter

The precise and reliable steps your QA team provides should hopefully enable you to reproduce the bug quickly. Make sure you understand what each step entails and follow their instructions as closely as possible.

You may be able to save some time by using tools (see “Build Your Own Tools” on [page 146](#)) or shortcuts built into your program, jumping over some reproduction steps to get directly to the spot where the bug occurs. If you can’t reproduce the bug anymore when skipping a specific step, you should avoid skipping it and try to understand how that step contributes to the bug.

IN PRACTICE: SKIP TIME-CONSUMING STEPS

I once worked on a tool that processed large numbers of assets, converting files from one format into another. The tool repeatedly crashed on one file after it had already spent hours processing hundreds or thousands of others. In this case, instead of waiting for hours before being able to investigate the issue, I spent some time exploring how to identify the problematic file and feed just that file into the tool. This drastically shortened my iteration times, and I was able to fix the bug soon after. (Reducing iteration times will be discussed in detail later.)

Pay attention to detail as you attempt to reproduce the steps described in the bug report. As mentioned earlier, a video recording or screenshot of the bug helps tremendously. Imitate the bug reporter’s hardware configuration as closely as possible—the same amount of RAM, same graphics card, and so on—and use the same version of the operating system or firmware. Matching steps also includes ensuring that you have identical drivers, runtime, input method, resolution, and display. Keep track of other programs running in the background that may interfere with the program and compare the amount of free disk space left.

If you’ve tried all this but still haven’t been able to reproduce the bug, start from the beginning and do everything exactly as described in the bug report. Complete all the steps in one sitting, from a fresh install up to the

point where the bug is supposed to occur, including potential delays described in the reproduction (for example, the bug might have occurred only after the program was left idle for a few hours). If the steps still don't yield the intended results, you may want to ask the QA team to find a more reliable reproduction method. Consider this only after closely following the steps and trying some variations of your own. It's easy to miss subtleties in a reporter's instructions and end up hitting a wall even when the described steps are correct.

Reproducing the issue in a different place or situation may help if the QA team is unable to give you more reliable instructions. Bugs that appear when a feature is used in different situations are likely restricted to that particular feature rather than coming from interactions with other systems. If you still can't reproduce the bug yourself, but the original reporter can with ease, let them do it for you. First, they should show you their procedure. You may spot differences in the reproduction steps or even the bug's cause. If that doesn't help, ask the reporter to follow the steps on the device you're using for debugging. The reporter may be pressing the buttons in a way you can't emulate or may perform an extra step they forgot to include in their report. In addition, the original reporter may be able to complete time-consuming reproductions more quickly than you. Once you've successfully reproduced the bug, add your findings to the report, including whether anything in your reproduction differs from the original situation.

Introduce Variations

Now that you've reproduced the bug, you can play with it by trying different variations of the reproduction steps. Your goal is to identify unnecessary steps, learn about interdependencies, and ideally find the one action responsible for the bug.

How much time you put into this part depends on the bug's elusiveness. If you're able to reproduce it easily and have a promising lead on the cause, you don't have to try any variations. But if you have a hard time tracking down anything related to the bug's symptoms in your program, the following approaches may be useful.

Skip some reproduction steps to find out which are essential to the bug. It's common for reporters to associate certain actions with a bug because

they were executing them when it showed up, even if they aren't actually necessary for reproduction. Be sure to remove any unneeded steps from the instructions.

Change the order of actions to learn more about the interdependencies between them. One task alone may put the program in a state where the bug will occur, while all the others just bring you to the place where you can execute it. Alternatively, multiple steps may build upon each other to trigger the bug, like when one modifies a state on which the next bases its logic.

Vary the timing between the steps to see whether this plays a role in the reproduction. If time has an influence, this further suggests that interdependencies are a source of the problem.

Try to trigger the bug in different areas of the program. If the reported bug occurs in the GUI, for example, try it in a different dialog or form. You can also change the environment by altering what else is loaded at the same time.

TIP: VARY THE REPRO STEPS

Suppose you encounter a web application bug in which putting text into the search field only randomly yields results. In this case, you might test different ways to enter the text (such as by copy-pasting or varying your typing speed). You could also try to reproduce the bug on a different page if the same search field is reused in multiple places. Additionally, you could vary how quickly you start using the search field after loading the page. Going through these variations should help you determine what makes the bug appear more often and what makes it disappear.

You can also completely avoid the action that makes the bug occur. Is the program running perfectly otherwise, or does another symptom emerge? For example, the bug could disappear in the step you test, then appear in a different section of the program or affect functionality that previously worked without issue. If there are no other problems, there's likely something wrong with this one part of the process. However, it's still possible that some internal state is corrupt but only manifests later. If there's a new symptom, it seems more likely that one of the previous actions is to blame for putting the program in a bad state, and the last step just reveals the issue.

All these findings set you up to look for more information using the strategies described in the next part. Just isolating the one step to blame might already give you a good idea of where the bug comes from. At least it gives you a spot to start investigating more deeply.

Beware of Networking Complexity

Bugs that involve only one instance of your program are usually easier to reproduce than those that show up in a networked context involving multiple programs or instances of the same program. You may need a group of people or several devices to reproduce the latter. To start, though, investigate whether you can reduce this added complexity to simplify reproduction.

First, confirm that this bug can be reproduced only in a networked scenario. Try reproducing it with just one instance of the program, then introduce some variations that might make the software operate more like it does in a networked setting.

If there's no way around operating within a network, check whether you can reproduce the bug without another person's help through methods like triggering multiple instances on a local machine or connecting a second machine and leaving it running while you control the local one. You could also ask a colleague to launch the program and work on something else while it runs in the background.

The more instances of the program that must be connected to each other, and the more users that must actively provide input, the more complex the reproduction becomes. In the worst case, you'll need multiple human users to control the program in a certain way and time their actions properly. If a bug that requires this isn't easily fixed, investigate whether you can remove this complexity by automating at least some parts of the reproduction (see "Automate the Reproduction" on [page 29](#)).

If automation isn't possible, you must make sure not to waste other people's time when they're working on the reproduction. Have a plan ready, ideally in written form, and distribute it to the team along with the original bug report. Answer the following questions:

- Which variations of the original reproduction steps do you want to try?
- If you have trouble, do you want to try alternative reproductions?

- If you're already at the fixing stage, what solutions do you have lined up?

The more complex the bug, the more critical it becomes for everyone to be on the same page.

2. Shorten the Feedback Loop

How? Reduce the time it takes you to reproduce your bug after modifying your program. You can decrease the time needed to build the code and the assets, the time it takes to deploy them to the target device, or the time your program needs to start.

Why? This allows you to make and test changes more quickly. We can subdivide the time between making a change and reproducing the bug into four optimizable steps, covered in the following sections.

Reduce Build Times

The first step you can optimize is building your code and assets. You can usually reduce at least some of the time this takes, but how much depends on the program and your development choices.

Code building strategies depend on the programming language you're using. For languages compiled to native executables, such as C++, C, and Rust, compilation can take a while. Others, like C# and Java, run on virtual machines and often compile more quickly. Lastly, interpreted languages like JavaScript, Lua, PHP, and Python don't need to be compiled at all.

Assets are data your program depends on to fulfill its intended purpose. GUI applications have assets like forms, images, and configuration files. Server applications require static files, templates, and databases. Games depend on textures, 3D meshes, levels, shaders, and so on. Some of these assets can be used as is, but others must be preprocessed or built, which can take significant time.

Accelerate Building Code

If you're using a language that must be compiled, you can sustainably reduce the time it takes to build your code by improving your project setup and choosing the right build tools. The steps this requires heavily depend on

the programming language and technologies you use. This section covers several insights into speeding up the compilation of C++ projects, most of which also apply to programs written in C and partially to other related languages. Though these languages are used for many large projects, they are notorious for their extensive build times.

Unity builds, also known as *amalgamated builds* or *jumbo builds*, allow you to build more quickly by compiling several source files as one compilation unit. Stuffing multiple *.cpp* files into one removes the need to parse repeatedly included header files for each. The build may break because of naming conflicts, but that's usually quickly resolved by changing one of the conflicting names or introducing namespaces.

To manually set up a unity build, you can create a new source file that includes the others by using the `#include` directive, then compile just that one. Build systems such as Visual Studio, FASTBuild, and CMake automate this process, while larger projects like Unreal Engine have custom systems that default to unity builds.

You can also distribute building across multiple machines in a network. If one developer needs to compile something, the build system can automatically share the workload with other machines. Incredibuild offers this solution, as do the open source projects Bazel and distcc. This technique can reduce build times drastically but is a relatively complex undertaking and works well only with machines in the same physical location, so it's not the most widely available feature.

Distributed caching is a variation on distributing building across machines that share previously produced build artifacts. As several developers will most likely end up compiling the same code, the resulting object files should be the same. If you have a central cache for these files, developers can share build results so compilation changes to a simple copy operation. Bazel supports this, but you can also use other tools like Ccache.

Precompiled header files are another widely available method for speeding up builds. Create one header file that includes the basic functionality you'll need in most of your compilation units, then let the compiler precompile it. Subsequent compilations of other files directly use the precompiled results. All popular compilers like Clang, GCC, and MSVC support this feature.

If none of these tools help, or if they aren't available to you, you may need to modify your code to compile faster. The size and number of included header files have the most significant influence on C++ compilation speed, so only use them to define and declare what is essential to your public interface. Tools such as include-what-you-use can help you determine which files are sufficient for a source file to compile and remove the unneeded includes.

TIP: USE IDIOMS LIKE PIMPL

You can often use just a forward declaration for classes because primarily pointers or references are passed around, and you only need to make the class definition available if another piece of code directly interacts with a member function or variable. To do so, you can use idioms like *pointer-to-implementation (PIMPL)*, a C++ design pattern that hides the implementation details of a class behind a pointer. The public interface remains unaffected when implementation details change, promoting flexibility and facilitating binary compatibility between different versions of the class. Here's how to implement this pattern for the function `DoSomething()` inside a class `MyClass`:

```
// .h file
class MyClass {
public:
    MyClass();
    ~MyClass();
    void DoSomething();
private:
    class Impl* pImpl;
};

// .cpp file
#include "MyClass.h"
class MyClass::Impl {
public:
    void DoSomething() {
        // Implementation details
    }
};

MyClass::MyClass() : pImpl (new Impl()) {}
MyClass::~~MyClass() {delete pImpl;}
void MyClass::DoSomething() {pImpl->DoSomething();}
```

The PIMPL idiom speeds up compilation when changing a function's implementation or internally used member variables. However, tweaking the class's public interface compilation will take just as long as if you hadn't used PIMPL. Alternatively, you can build a bare-bones base class and provide further implementation details in a derived class in a source file to get more control over public exposure.

It's difficult to move function implementations to source files when it comes to templates, as instantiating one requires the whole template definition. One way to deal with this is to simply use fewer templates and rely on other techniques, which often come with fewer type-safety guarantees (or none, in the case of passing `void` pointers around). Since C++11, `extern template` allows the reuse of template instantiations from other translation units, but this feature also has its caveats, such as limited support and the overhead of managing which templates are instantiated where. Be sure you have a clear strategy when using it. C++20's modules may also improve compilation time if you use them carefully. Be mindful of the optimization options you use to compile your project. These options have a lot of influence on the build times. Very high optimization levels (like `o3`) or link-time optimization (LTO) typically take a lot longer than normal optimization levels (like `o2`). Therefore, it's a good strategy to use the highest optimization levels for end user or QA builds and more average levels when building locally. (However, differences in your program may cause bugs in fully optimized builds that you can't reproduce with lower optimization levels. The highest level, `o3`, is known to apply some strategies that may break your code, especially if it doesn't comply fully with the C++ specification.)

Avoid Long Compile Times

If you modify just one or two source files while you're investigating the bug, it usually won't take long to compile the changes, since incremental compilation is excellent nowadays.

Long compile times come from having to modify header files. In the worst case, one tiny header file modification invokes a rebuild of the whole codebase. If not that, the number of files to rebuild may still slow you down significantly.

When iterating on a fix or feature, to avoid making changes to header files, manually put your declarations in the source files where they are needed. This sort of small-scale change is a bit hacky but can speed up your inquiry. For example, suppose you need to introduce a new function defined in one source file and called from another. You can put the definition where it belongs, then just declare the function in the source file where you need to call it. Likewise, if you need a global variable to share some data between different files, define it in one and declare it with `extern` in the others.

Such a global variable may even work well if you want to try something quickly without implementing it correctly. If your codebase has a relatively clean design, there are most likely standard ways of handling dependencies between various systems and their communication. For a simple experiment while fixing a bug, use a global variable without forwarding the data dependency in all the right ways.

It's crucial to correct such a hacky change before checking it into the version control system. Marking it with a special comment that doesn't exist yet in the codebase is helpful, since it allows you to easily search for and fix the change. That comment can be as simple as `// <YOURNAME>` or, as one of my colleagues preferred, `// stupid`.

Accelerate Building Assets

Some programs use modified assets without any preprocessing, such as an XML file describing a window layout or a PNG showing a logo. Technical constraints like performance or memory requirements may mean other programs need preprocessed assets. (For example, parsing an XML file can take quite a bit of memory and time.) This subsection discusses quick ways to build assets that require preprocessing.

First, carefully consider which parts of your asset pipeline are time consuming, reviewing the timing of individual steps or profiling the pipeline in greater detail. Look into the features your framework provides to speed up building. As with code, you may be able to distribute a build across multiple machines or cache build artifacts in a central location. It can also be an excellent idea to further parallelize asset processing in case some of your CPU cores are not yet fully utilized.

You may be able to avoid building all your assets before launching the program and instead either implement or leverage functionality where the program requests assets when needed, then processes them on demand. However, there's a trade-off between runtime performance and the time needed to preprocess the assets.

For development builds, it can be helpful to reduce the quality of the processed assets. In particular, texture compression frequently becomes a bottleneck. Reducing the texture resolution and using lower-quality compression looks much worse, but speeds up processing in the asset pipeline, transferring the processed textures if needed and loading them at runtime. Likewise, if you preprocess your 3D meshes, you can do so at a lower quality to make generation faster.

Shaders can also block progress, usually due to the sheer number that a program must compile (rather than a single shader's compilation time). Many 3D frameworks opt to generate shader permutations by hard-coding features such as the number of light sources or the presence of certain features, which could result in more than a hundred thousand shaders per project.

If all these shaders are needed, there's little you can do to improve iteration time. Specific bugs or features, however, probably don't require all of them. For example, good dependency tracking might let you use only the shaders for a specific scene. You could also disable rendering features like ray tracing if the bug is unrelated, or just let the program request shaders when needed.

Reduce Deployment Times

The next step to consider optimizing is the deployment of any code or assets you've modified and built. This step can be relatively straightforward (or nonexistent) if your program doesn't depend on a lot of data and you're running the build locally on the PC you use for development. However, deployment becomes a huge time drain for large programs running on remote devices. This subsection discusses ways to speed up deploying your modifications in such cases.

Choose a Fast Deployment Solution

Target platforms other than PCs, such as game consoles, development kits, and mobile devices, require you to find suitable deployment methods. Method choice is also important for networked programs with remote servers or multiple clients that need updates. Games, for example, may need you to deploy tens of gigabytes of content before addressing a bug. In such cases, any optimization to reduce deployment times helps, no matter how small.

The tooling for some devices allows the deployed program to transparently access your computer's files. Depending on the connection and the overhead of these file requests, this may be relatively slow, but it's advantageous if the program just requests certain files it needs. In this case, you may be able to leave out lots of data that would have been transferred otherwise.

Other devices may force you to copy over your changes every time. This won't take long if the changed artifact is small and atomically replaced, but if you can replace only bigger chunks at once, it may be time-consuming. Such problems typically arise if you compress all assets into one large archive of multiple gigabytes, which then has to be copied over. Some platforms and frameworks support overlaying individual assets on top of existing installations or archives. This is a great solution, since you can surgically replace what you've changed.

If the platform's tooling doesn't support transferring files on demand, you can implement this capability yourself. With a somewhat sophisticated file access abstraction, you can add another file system implementation that sends out network requests to a file server, transferring the data and caching it locally for further action. Some implementations do this via regular HTTP requests, but more specialized protocols can yield better performance.

Whether you're copying everything up front or selecting individual files, the connection speed between your development device and target can slow the process down. To fix this, look into the hardware setup. If you have a USB connection, use a cable and PC port that allow for the fastest available transfer speeds. If you're on Ethernet, never use 100MBps; you may be able to work with 1GBps, but network setups that support 10GBps are possible with some development hardware. If you must rely on Wi-Fi to transfer data, make sure you have a stable connection, are in close

proximity to the access point, and are using one of the newer, faster standards.

Deploy Less

Another way to shorten the time for deploying changes is to reduce what needs copying. For example, if you're iterating on a bug in a specific game level, you can cut down your package to include just that level and its dependencies. Alternatively, with the bug isolated to a particular enemy character, you could eliminate many dependencies by creating a level with only that enemy. You can also reduce the size of the data itself by limiting the resolution of textures or video files or cutting whole subsystems such as sound.

However, you must be careful not to accidentally remove something that influences the bug. If the bug is timing-related, for instance, reducing the load times of assets could magically make it vanish. If you change the environment, always ensure that the bug is still reproducible and behaves the same way. You'll need reliable asset dependency tracking to make some of these improvements. Ideally, your framework should support only packaging or deploying referenced assets and their dependencies, such as bringing on textures, skeletons, and meshes for a 3D model your program requires.

Dependency tracking avoids quite a few bugs that might come from missing assets while reducing distribution size. It might even reduce expenses down the road if you're distributing the data to users digitally or putting the program on physical media.

Leverage Hot Reloading

Another technique for minimizing iteration times is *hot reloading*, loading new code or assets at runtime without having to restart the program. It's worth investigating whether your technology stack and your target platform support hot reloading, since an increasing number of platforms offer it.

Code is easier to reload in some use cases than in others. If you implement logic in a scripting language such as Python, it's relatively straightforward to allow hot reloading for changed scripts; you can provide all the necessary infrastructure yourself. (Be aware that structural changes and new logic could lose or reset data, though, and that maintaining hooks

to and from native code is tough.) Hot reloading is also possible with code running inside virtual machines, as with C#, but becomes more complex with native languages like C or C++. The latter is especially tricky because it has many more language features than C, allowing for more complex logic and data dependencies.

For a native, compiled language running on a bare-metal machine, your system must support hot reloading in some way. You can often achieve this by reloading a dynamic link library (a *.dll* or *.so* file), but you can also hot-reload the executable itself. You may need to implement dedicated code to patch up changes in the data on which the new logic depends so you can format it correctly. Different solutions offer varying levels of reliability when hot reloading. Visual Studio, for example, has a feature called Edit and Continue that offers hot reloading while debugging. You can also use software dedicated to hot reloading, such as Live++.

When iterating on a bug, ensure that your hot reloading works correctly and doesn't cause problems that have nothing to do with the bug itself. Hot reloading often isn't well-tested because it's purely a development tool. It's prone to its own bugs, which could sidetrack you from investigating the bug you were originally working on. Make sure to restart with a clean state if hot reloading seems to be the culprit. The availability of hot reloading for assets depends on whether your framework supports swapping them out at runtime without requiring a restart. Consider implementing such a feature if using your own custom framework. This can be a simple process if executed with sufficient foresight. Ideally, the feature should be part of the core architecture because many low-level systems need to allow for asset data to change at specific times and higher-level systems may need to react to modified data. Dependencies complicate reloading, requiring more systems to be notified and react accordingly, potentially at unfortunate times when they are busy with other things.

IN PRACTICE: HANDLE DEPENDENCIES DURING HOT RELOADING

I once had to add hot reloading to our bespoke 3D renderer. The first step, handling purely visual assets like shaders and textures, was easy. Of course, re-creating them when the file changed required new logic as well as an indirection enabling dependents to refer to the asset even after internal re-creation, but no other logic had to react.

However, things got more complicated in cases where dependents had to be notified. For example, some features required CPU-side logic to examine textures and therefore needed a system that registered callbacks and distributed notifications to run with the updated texture data.

Changes to the format of textures, meshes, or shaders also had more complex implications. The non-trivial task of re-creating texture samplers, pipeline state objects, or material bindings required the propagation of such updates to other components internal to the renderer.

Reduce Startup Times

If you can't use hot reloading or the change isn't compatible with it, you'll have to boot the program from scratch after your changes have been deployed. This step covers the time between launching the program and being prepared to start acting on the reproduction steps in the bug report.

Remove Unneeded Steps

To minimize startup times, think of everything that gets in the way. Chiefly focus on steps you must perform manually and those that make you wait. Remove anything that takes a substantial amount of time and isn't strictly required, such as legal or splash screens (showing copyrights and so on) displayed on startup.

In addition, while some startup screens load data that's necessary to continue, others preload data needed only for specific features. This preloading typically avoids loading on demand later so that users don't have to wait, but you can probably skip it if your bug isn't related to or dependent on preloaded data.

Removing menus can also make a big difference by letting you avoid user input. With the menus present, your workflow might look something like this:

1. Launch the program from the debugger.
2. Wait for the program's launch and the introductory screens and videos to pass.
3. Wait for the initial data to finish loading.
4. Navigate the menus.
5. Wait for more loading.

6. Try to reproduce the bug.

Anytime you switch between waiting and doing something, you'll lose time, since you won't be able to press the right button instantly.

Worse, you might get distracted and start looking into something else while waiting. Reading up on another bug, writing documentation, or scrolling through your friends' latest achievements on social media are irrelevant to the bug at hand. Context switching could make you forget anything from a new lead you were considering to how to execute a reproduction step. Strive to remove all opportunities for distraction, predominantly by ruling out operations you have to wait on.

In addition to reducing the number of steps, you should also investigate whether you can reduce the duration individual steps take. For example, if the step is to type 100 characters, you can simply speed up your typing. For other steps, it may be possible to repeat an action just 5 times instead of 10. It may also be viable to wait for a shorter period than the instructions suggest or use a prefilled template instead of creating a new document and filling it with data. When tweaking a step, make sure you can still reproduce the bug just as the step's original instructions did.

Optimize Loading

After reducing the steps required for startup, spend some time profiling and optimizing the loading process itself. This is beneficial not just during development but also to end users when the program ships. Each of the following steps can potentially be optimized:

1. Read the data from the disk into memory.
2. Potentially decompress the data.
3. Identify dependencies and trigger loading them.
4. Process the data to prepare it for its intended use case.
5. In the case of graphics data, copy the data to video memory.

Aside from optimizing the data-loading process, you can also reduce the amount of data that must be loaded, as we discussed earlier.

Additionally, you can speed up reading the data by using faster hardware, like a fast SSD instead of an HDD, or combine small reads into

one big one to reduce the overhead of API calls and allow the system and hardware to optimize for one large data transfer.

Different platforms may let you use speedier APIs. Asynchronous IO APIs, like Overlapped IO for Windows or the newer DirectStorage, tend to be faster because they allow the underlying operating system to schedule individual operations in a more optimal fashion. Memory mapping your data (such as by calling the `mmap()` function) might allow the operating system to speed up your reads. However, make sure to profile such a change; you may in turn be able to outperform the OS if you know more about how the data should be handled than it does.

If your framework doesn't compress its data files by default, consider enabling or implementing this feature. The benefit of reading less data from disk almost always outweighs the time decompression takes. While this means trading one resource (disk IO) for another (CPU time), it can greatly improve loading speeds.

You can also optimize decompression on the CPU. One approach involves parallelizing it and executing it on the background thread, which is often a free resource, since CPUs today tend to have many unused cores. You can even look into spreading decompression across multiple threads. Depending on the compression algorithm, you may be able to perform one decompression process in parallel. If that's not possible, you can still parallelize multiple processes by handling multiple files in parallel or splitting your files into chunks to be decompressed individually.

Another way to improve decompression is to choose a faster compression algorithm. Closely compare the different parameters, such as compression speed, decompression speed, and compression ratio. Generally, you should get faster decompression speeds with a compression ratio similar to the slower algorithm. Compression may take longer, but that shouldn't be a problem, as it happens offline and can be done on powerful hardware.

The default choice for algorithms is often zlib (<https://www.zlib.net>), but other options are much faster. One popular alternative is Zstandard (<https://github.com/facebook/zstd>), which is free and offers excellent decompression speeds. If you're able to look into commercial offerings, Oodle (<https://www.radgametools.com/oodle.htm>) is even better. Use any

hardware decompression that the target platform supports, as it will be much faster than the software alternative.

Many assets have dependencies. For instance, a 3D scene depends on all the meshes it contains. A mesh, in turn, depends on materials that must be properly rendered with shaders, textures, and so on. A large scene requires handling many dependencies. When a program requests one asset, it may turn out to need hundreds. Optimize your framework for handling dependencies by identifying and starting to process them as early as possible.

Processing the data loaded into memory depends on the asset type at hand. You must deserialize some data, such as a process for which many different approaches exist. Some libraries advertise allowing zero-copy deserialization, in which the read data is used directly as a `struct` or `class` without further processing. The ideal solution would be to avoid processing anything, but if you're optimizing data for the target machine, this implies that the serialization step has to take said target machine into account, meaning your data won't be usable on all target platforms.

Different types of assets may require other kinds of further processing. For example, for 3D geometry, you may need to create a physics representation. Audio data must be passed on to the audio system to create the right buffers, and shaders may have to compile. Evaluate how to speed up this special processing on a case-by-case basis.

Jump to the Correct Location

Once you've made it through the previous steps—built your changes, deployed them, and started up the program—you're ready to start reproduction. Depending on your bug, you're probably inside the program proper at this point, but not in the right screen, menu, level, or component. You need to get to the correct location as efficiently as possible. The best solution is often to automate this process, which we'll discuss in the following section. Alternatively, you can use integrated tools that let you jump to the right place (see “Build Your Own Tools” on [page 146](#)).

3. Automate the Reproduction

How? Automating the reproduction can take many different forms. You can configure specific settings, so your program does what you want on startup. You can hardcode or use scripts to execute the reproduction steps. You can also implement recorded inputs or a serialized program state. Lastly, you can modify your program to run only the logic that exhibits the bug.

Why? An automated reproduction frees up your time, and you can do something else while the computer works. Automation is especially beneficial for bugs that occur infrequently, require exact procedures, or have many long reproduction steps. Whether you automate individual steps, executing the broken functionality, or basic tasks like starting a program, the decision can potentially save you significant time both in reproduction and in fix testing. This section discusses when automation makes sense, then presents several methods of implementing it.

Whether to Automate

If setting up complete or partial automation won't take much effort, it's almost certainly worth doing. It's hard to predict how much time and work automation will require compared to manual reproduction, so you'll have to make a judgment call.

Some reproduction processes require many individual steps or steps that take a long time, such as using 50 features one after another or processing a large dataset. Automating these steps can let you run them while you do other things. This is also true for rarely occurring problems and situations where getting the reproduction right takes many attempts, as with bugs that occur only when you do something at a specific time or in a specific way. Even if there's a random factor to the reproduction, the computer is happier than you are to try again and again until the bug finally happens.

The computer may be able to reproduce the bug faster than you can because it knows exactly when it can start the next step or because it can run the actions faster. It might even be able to take shortcuts, such as running the code to use 50 different features directly instead of selecting them individually from the menus. Greater precision may also be possible: If you must click in the exact right position when triggering these 50

features, a manual reproduction would require fine-tuning using tiny adjustments to avoid mistakes, while the computer could instantly get it right every time.

Finally, an automated reproduction scales easily. You can leave it running overnight, over the weekend, or for days on end if you have a spare machine to dedicate to the task. You can even run the reproduction on several devices to drive up the chances of quickly reproducing the bug.

Make sure your proposed automated reproduction can notify you when the bug has occurred so you can start to analyze it, ideally by connecting a debugger to the machine on which the program is running. If you can't use a debugger, you must at least be able to retrieve log files or crash dumps from the device (see “Collect Crash Dumps” on [page 91](#) and “Debug with `printf`” on [page 158](#)). If you reproduce the bug but have no way of looking into what happened, you've gained nothing.

Even with these advantages, automation might not be worth the hassle if it'll take too much work. The next section discusses various methods of implementing partial and complete automation, which may help determine how much effort automating a particular reproduction process would be.

How to Automate

Let's start with the most basic type of automation: You should be able to configure your program's startup process to automatically find the place where the bug occurs and to pass screens, menus, or components to open as command line arguments or in configuration files. Unfortunately, features that put your program in the right place don't get you further in the reproduction, because the following steps presumably require actual interaction with the program. The subsequent sections describe various ways to automate that interaction, which your framework will already facilitate if you're lucky.

Program the Reproduction Steps

One way to implement interactive reproduction steps is to hardcode them. You can call the respective functions that user inputs would otherwise trigger or set up fake user inputs to indirectly cause them. You can also resort to using integrated tools (see “Build Your Own Tools” on [page 146](#)).

After all, if they were intended to simplify reproduction by a human, they should also be able to simplify it through a computer.

Hardcoding reproduction might be more challenging than you expect. Reproduction steps in a bug report can summarize many individual actions that are trivial for a user to execute but difficult for a programmer to implement. For example, a request like “paint a new image containing at least 4,000 brush strokes” requires many individual actions for either a human or a computer if the goal is to simulate the user as closely as possible. It can even be complex to program an instruction like “move the cursor to the end of the line” step by step, since it requires continuous input until the task is done. Hardcoding requires a system that can direct and orchestrate all these steps and track progress over a longer duration. That includes delays, waiting for feedback, moving around (not necessarily in straight lines), and so on.

IN PRACTICE: REPURPOSE EXISTING FUNCTIONALITY

A game I worked on repurposed existing artificial intelligence (AI) systems to implement an automated character that could play the game to some extent without player inputs.

The team used this functionality to discover bugs and reproduce them by providing the AI with an ideal path through the game world and triggers to achieve more complex goals when it reached specific points. As a result, the AI could fight groups of enemies and solve simple puzzles. It wasn't the smartest character, but it didn't matter: As long as the AI could traverse levels and play missions, it could help reproduce bugs.

Setting up reproduction in a program's source code requires compiling the code, deploying the executable, and all the other tasks that come with code changes, but gives you immense flexibility and power. There's no functionality you can't access and no layer between the reproduction and program code that might influence or prevent the reproduction.

Alternatively, you can implement the reproduction steps in an external script. This technique offers better turnaround times compared to hardcoding, but that extra layer (or layers) between the program and the reproduction may influence whether you can successfully reproduce the bug. You also must expose everything that needs to be called to the script,

which can involve extra work if you wouldn't need the functionality exposed for non-debugging purposes.

Record Inputs

A more general approach than programming specific reproduction steps is using a tool that lets you record and play back user input. This method works best with a deterministic program, one that behaves consistently every time (see “Remove Randomness” on [page 69](#) for details on how to make your program deterministic).

Be mindful when changing the program. If you alter anything about its interface or behavior, it can invalidate the previously recorded inputs. Either don't make changes that could affect compatibility or make sure you update the recordings once you do. You can implement the input recording and replay either internally or externally to the program and framework. With an internal implementation, the recording and playback hook directly into the framework's input system, which you can customize to suit your use case. You'll have to make sure the program exactly replicates user input.

Alternatively, you can automate recording and playback on the operating system level. In this approach, the recording intercepts and stores arbitrary user input events before forwarding them to the program, at which point the playback inserts them back into the OS's event handling. This has two advantages. First, you can use an external system for multiple programs. Second, if you can implement playback on the event level where inputs usually occur, the program won't distinguish simulated events from those that actual users cause.

Keyboards, mice, gamepads, and other devices can produce a wide variety of input events, which you'll need to consider during automation. Boolean buttons like keyboard keys and mouse buttons generally support two states, on or off. Continuous axes, often represented with a floating-point number, are more complicated to deal with because different hardware devices and input methods provide varying ranges. Recorded values may not easily transfer between different devices. Some gamepads output an integer value between 0 and 255 for the triggers; others report values between 0.0 and 1.0, or even -1.0 and 1.0 . Gyroscope and acceleration sensors also offer a variety of values.

Remember that devices can be connected and disconnected while a program is running. The recording implementation should replicate these events, as handling disconnects, for example, may influence your ability to reproduce a bug. While keyboards and mice rarely require special handling, devices such as tablets or gamepads may need distinct implementations.

Multiple events from the same device, or even for the same button, may appear in a very short time frame. Some buttons can trigger only instantaneous events, while other mechanisms, such as mouse wheels, can cause multiple events simultaneously. The user may simply perform their actions very quickly, so the pressed state lasts less time than the program expects, and it reports both up and down events the next time it handles inputs.

Focus on Specific Implementations

So far, these automation methods have focused on bugs requiring user-like inputs. However, if you've isolated where the bug occurs, your automation can specifically address the problematic functions or systems. The solution can be as simple as calling a function with specific arguments. In a more complex scenario, you may have to set up other systems before executing the one exhibiting the bug.

Call into the problematic functionality as early as possible during the program's startup. Once all dependencies are up and running, it's time to trigger the bug. If there are no dependencies, call the buggy code directly as the first statement in the `main()` function. This reduces your iteration time by minimizing the duration between launching the program and getting to the spot where the bug occurs.

IN PRACTICE: TEST FUNCTIONS EARLY ON

Sometimes, you just need to test a specific function with a few different inputs. I once had to debug a function that combined filepath segments to form a complete path. This seemed like a relatively simple task. However, since a path potentially involves two separators (`/` and `\`), it can be tricky to ensure there's always exactly one separator between two elements, independent of whether the inputs to the function contain them.

To investigate this, I put a few calls to this function at the beginning of my program into `main()`. I knew the arguments I provided would reproduce the issue, but I tried to cover some more cases by providing arguments that I thought could be problematic,

too. I was able to identify a few bugs that would otherwise have taken a long time to address when someone accidentally stumbled upon one.

If the bug may depend on specific inputs to a function, calling it early on startup with varying values is a great way to find which values are problematic. You can easily change them and restart the program without much overhead, even putting the function call in a loop that iterates over a range of inputs to find the offending ones. If the code fails randomly, call it repeatedly in a loop until it fails. When the randomness is due to threading or timing, try calling the function at different times or repeatedly to maximize the chance of triggering the bug.

Exposing function calls like this is a great way to determine the preconditions for bugs that depend on a certain context. You can easily change what other functionality is called beforehand, in what order, and with what parameters, and you should do so in a structured way until you discover what makes your buggy code fail. Keep notes about what configurations you have tried so you don't duplicate your own efforts (see “Keep a Debugging Log” on [page 140](#)).

If the reproduction instead depends on a certain asset being broken, you can change your loading code to always load the offending asset even when not specifically requested. Just be careful not to replace all assets with that one, as that might break other things.

Facilitate Immediate Reproduction

A reproduction generally depends on the program being in a particular state when the user performs a specific action. If you can save this state in some way and restore it, it's easier to take the action.

Ideally, you could save the complete state of your program without further work, but that's often not easily done. Some programs already offer to serialize projects, contexts, and any progress and save the results to disk. Depending on the programming language, this might be an excellent basis for adding additional state information.

Loading a saved state of the program just before the bug occurs may take you precisely to the right spot, provide all the needed data, and execute all necessary previous actions. Loading the state directly reproduces the

issue, or you can easily reproduce it manually from there with very few further steps. This way, you can drastically reduce the time required for reproduction.

If your program supports this functionality, you should ask the reporter for a saved state that reproduces their bug with minimal effort. Alternatively, you can reproduce the bug yourself and save the state to reduce the time for subsequent iterations.

IN PRACTICE: USE SAVED DATA AS A SHORTCUT

A tool I once developed consisted of two different executables. One of them collected all files that needed to be processed, then generated a parameters file that it passed to the second executable. The parameters file contained all the necessary information that the second executable needed to do its work. Every now and then, one of these files caused issues in the second executable. Getting a convenient reproduction was easy: I just saved the parameters file and called the second executable manually to produce the bug.

As another example, I once worked on a game where purchasing a particular clothing item in a specific store caused a bug. The reproduction steps told me to unlock the store in-game and earn enough money to buy the clothing, which was a lot of work at first. Once I got there, though, I saved the game right after fulfilling the requirements. Every reproduction afterward was much shorter.

Leveraging a serialized state may lead you down the wrong path, though, if the state itself is broken. A previous bug somewhere may have caused the state to contain incorrect information, leading to a bug that has nothing to do with the one you're investigating. You may also get sidetracked because the serialized state doesn't truly reproduce the bug's conditions or make the reproduction possible.

When You Can't Automate

If you can't automate reproduction for the bug you're working on, that suggests the need for a human component. Some particular timing between actions, varying movements, or other behavior might be the culprit behind the bug.

Missing steps can also keep you from automating reproduction. A few of the suggested automation approaches rely on skipping actions or taking

shortcuts that wouldn't be available to regular users. Even if doing so does prevent reproduction, that provides you with useful information, as you can assume the actions you don't take are important, and you need to consider them when looking for a fix.

Summary

At this point, you should have a bug you can reproduce reliably enough to study, fast enough to iterate on, and ideally automated so the computer can do the work while you do something else. That's the foundation everything else builds on. You can't investigate a bug you can't trigger, and you can't trust a fix you can't test. The next question becomes "Where is the bug hiding?" [Part II](#) offers strategies for probing your code, your data, and your tools to find an entry point worth investigating.

Part II

PROBE



Once you're able to reproduce your bug, the next step is to find an entry point: the spot in the program's sources that connects the symptoms you're seeing to the code that produced them. Source code is complex, and large programs have a lot of it. Indirections and intermediate steps can obscure the connection between code and program behavior. Feature names might differ between code and user interfaces, and they may evolve as the program's requirements change.

Not all bugs come from code, either. Insufficient checks in the development tooling may let buggy assets through. While most bugs can ultimately be blamed on a programmer in some way, other disciplines may also be responsible for the root causes. Hardware or third-party software that you don't have the source for can be the real culprit.

The strategies in this part will help you find an entry point quickly. Try a shortcut, search the code, search the internet, ask someone else, attach a debugger, or look at a crash dump.

4. Try Shortcuts

How? Based on your experience, you can make educated guesses about which areas of your program's source relate to your bug's symptoms. You can also check systems that have repeatedly had bugs and confirm they're not at fault, initially ignoring the ones with responsibilities that probably don't relate to your particular problem.

Why? Once you know you need to jump to a specific point in your program's sources, getting there doesn't take much time: With most integrated development environments (IDEs) or editors, it takes only a moment to jump from line 20 to line 10,438 in a file. One method of identifying the part you need is narrowing down the target area by ruling out larger sections. If you can disregard most of the program's sources, you'll have a better idea of where to look for your target line. The presented shortcuts can work either way: discovering that a system is related to your bug or ruling it out.

Shortcuts are alluring. It's difficult to resist skipping all the hard work and getting a task done a lot quicker than your team would typically expect. However, you must be careful not to fall into the trap of repeatedly trying to find shortcuts without making any progress. Likewise, it's easy to get too focused on one where the line of inquiry doesn't lead to the spot that's at fault. If shortcuts don't quickly get you where you need to go, they slow you down because they keep you from beginning detailed investigations. In the worst case, you're mistaken and spend time looking into a supposed lead that turns out to be a dead end.

Throughout your work, keep checking to make sure you're on the right track. Does the spot you found via a shortcut really fit the bug you're working on? Is it reasonable to conclude that this code area is related to the observed symptoms? Or might this code be unrelated, meaning you can rule out this area as the source of the problem? Make sure to stay focused.

IN PRACTICE: AVOID UNLIKELY CULPRITS

On one project, my colleague was working on a bug related to network communication: Something was going wrong with our HTTP requests. The project used curl, a library that implements several network protocols. My colleague followed

the bug's traces and spent a lot of time trying to debug curl itself because that seemed like the logical source of the problem.

However, libraries like curl have been developed over numerous years, creating highly optimized, complex code that can be hard to read and understand if you haven't dealt with it before. Ultimately, it's more likely that the bug is somewhere in the usage of the library rather than in the library itself. Indeed, my colleague's bug turned out to be hiding in our code.

Sometimes taking a shortcut is impossible. It requires some experience with software development in general and with the current project. The more experience and knowledge you gather in both areas, the more valuable this strategy will be.

If shortcuts don't lead to the entry point, you can still learn something from where they take you. You can ignore unrelated areas during the remainder of your investigation and employ other strategies on vaguely related systems. For example, if you find a function somehow related to the bug, you can use a debugger to put a breakpoint in it and drill down further from there (see "Step Through the Code" on [page 100](#)).

The following sections introduce several possible shortcut methods. They may very well fall short, but if they're successful, they'll let you jump directly to the right point without any deeper investigation.

Make Educated Guesses

You might start by making an educated guess about the problem's location. In a perfect situation, you would have encountered this exact bug before, but you can't rely on that every time. However, if you're familiar with the codebase, this particular kind of bug, or debugging in general, you may be able to make a confident guess. Let's consider a few sources of experience.

A Particular Program

If you've worked on a specific program for a long time, you'll get to know its idiosyncrasies and how it reacts to certain kinds of bugs. You may be able to determine a bug's root cause and location from its symptoms alone. This is more likely with bugs similar to those you've worked on before, but it's also possible for bugs that interact with systems you know.

Understanding the program's overall architecture can provide information about how it works in general. Once you start investigating a bug and getting to know its surroundings, connected systems, and related functionality, you'll begin building a mental model of its context, which makes it easier for you to navigate the program's sources and get to any relevant point quicker.

Knowing fundamental decisions from the program's development and details about its technical debt also provides insight into approaching a bug. If you're familiar with the program's long-lasting temporary workarounds or hacky solutions to previous bugs, you have some great starting points to investigate because you have good reason to be suspicious of them.

IN PRACTICE: QUESTION YOUR ASSUMPTIONS

I once had to fix a seemingly simple bug in a tool my organization used internally. The program wasn't super complex, the bug report was very clear, and the requested change didn't seem outrageously complicated. When I started debugging the tool, however, none of the breakpoints I had previously set up seemed to do anything.

Then I started investigating my development setup. Surely something must have been wrong with the IDE, or maybe I was using the wrong build configuration without debugging information? I spent a lot of time checking everything around the program, but I couldn't find any problems.

Eventually, I took a step back and examined what the program did during its launch. I discovered that the initial launch executable didn't do much. It just checked whether an update was available and then launched a different executable, which I wasn't debugging. This piece allowed me to simply launch the other executable and start work. From then on, I was a lot quicker at fixing bugs in this tool.

Common Functionality

In addition to knowing a particular program, if you've worked on similar functionality before, you probably know how it can break. For example, you might know what can cut off sounds, why GUI elements may not adjust properly when resizing a window, or why a command line tool might misinterpret a filepath with spaces in it. When a bug occurs in a comparable functionality, you probably have at least an inkling of what's going on and where to start looking, even if you don't have a solution ready. This experience gives you an advantage across different projects.

IN PRACTICE: DERIVE LEADS FROM INCORRECT BEHAVIOR

Whenever I see a blank space, the outline of a box, or even a box with a question mark inside replacing a character in the middle of text, I have a good idea about what's going on. Because I've seen this problem several times already, whether I'm working on a web application, desktop GUI software, or a 3D program that uses a bespoke implementation to render text, I know two facts for sure. First, that character must be out of the ordinary—that is, not ASCII. Second, the font used for rendering must not support the character. In addition, the function decoding the text might not handle special characters correctly. All this provides me with multiple avenues of investigation, one of which will likely lead to the bug's root cause.

Lots of functionality is based on the same ideas and implementation principles. Ultimately, if a piece of code fulfills the same purpose as another, it likely works similarly. The details contain many differences, but similar functionality can lead to similar bugs.

Widely Used Frameworks

You can apply your experience with established frameworks across multiple projects. Frameworks like Electron, React, and WinForms make developers' lives easier because they provide a basis to build programs. Knowing a program's framework tells you a lot: what restrictions it imposes, what pitfalls developers commonly fall into, how bugs tend to manifest, and what tools the framework provides for tracking them down. It also tells you the framework's own weaknesses, so you know where to look first when something breaks.

The significant differences in how frameworks are developed may influence how you approach a bug inside or connected to one. With the source code available to you, you can closely analyze it to determine why something breaks. If the framework developer doesn't share the source, it'll be impossible to fix a bug directly, but you can still use your knowledge to work around such an issue or ask the developer for a fix.

A company may develop bespoke frameworks to use internally and share across multiple projects. While knowing about those is of limited use at other organizations, you can use your experience when transferring internally. You may also be able to apply concepts you've learned from

bespoke frameworks to others because their implementation may be similar if the functionality is comparable.

Foundational APIs

Programs and frameworks with similar purposes are written in similar ways partly because their systems must interact with the same foundational APIs. Knowing these APIs and their potential problems allows you to avoid, identify, and fix related bugs.

A program must interact with the same operating systems for input handling, window creation, multithreading, and so on. It most likely uses the programming language's standard library for all kinds of purposes, such as containers, networking, and file access. If the software uses a database, it must interact with a well-established implementation such as InfluxDB, MongoDB, MySQL, or PostgreSQL. Programs that support rendering 3D scenes must use a graphics API like Direct3D or Vulkan.

While frameworks often abstract away the interaction with foundational APIs, their peculiarities and limitations can still leak through, exposing you to the same behavior and properties—and potentially the same bugs. If an API has particular challenges, developers will likely make the same mistakes or overlook the same unexpected activity when implementing the code that uses it. Complex pre- and post-conditions may also make an API really difficult.

IN PRACTICE: BEWARE OF AMBIGUOUS NAMES

A platform I once worked on offered an API to play audio samples. The function in question had the following signature:

```
void play_samples(void* data, size_t length, size_t sample_b  
its)
```

I expected the second parameter to specify the length of the data in bytes, which I happily provided. The result was an audio track that sounded good for a while but had a lot of noise toward the end.

It turned out that the API actually required me to provide the number of samples for the second parameter. While the number of samples relates to size in bytes, it's not the same in most cases. For a 16-bit sample, the size in bytes would be 2, but the sample count would be 1. That's why the audio initially sounded fine,

then played noise for the same duration. Admittedly, the documentation probably specified that `length` must be the sample count, but looking at the function's signature made it easy to jump to the wrong conclusion.

The next time I work on this platform and encounter a bug related to audio playback, I'll check this particular function first. Learning about an easily misused function like this one makes it easier to fix similar bugs in the future.

Common Patterns

In addition to programs being influenced by their interaction with common APIs, many have similar functionality, which leads to shared implementation patterns. (Some books even catalog and proliferate those designs.) Even if you don't intentionally use specific patterns, your architectural decisions are likely influenced by other software you've seen, the source code you've read, and any code you've written. Ultimately, you can match any designs you've seen before, especially those you've used, with your mental model to quickly understand a particular implementation. You'll also know some of the pattern's shortcomings and potential pitfalls that may cause bugs.

For example, the model-view-controller (MVC) pattern divides responsibilities based on the data, its display, and the logic applied to it. Most programs that have the terms `model`, `view`, and `controller` in the code use this design. Knowing that means you have a general idea of how the code is built and which bugs can potentially hide around its use, such as the way its relatively static dependency chain can make updating the view inefficient. That makes it a lot easier to find an entry point in the code for a bug where the view always lags.

IN PRACTICE: MIND INSTANTANEOUS INPUT EVENTS

I've often had to deal with missing user input, but that problem isn't based on a popular pattern. It comes from low-level input handling interfaces, which work in one of two ways. Either the program polls for the state of the relevant keys and sensors or it listens to events that the operating system generates. Bugs that lose input must be connected to misusing one of these paradigms and usually result from brief holding time or quick repetition.

If the code polls the input state periodically, it's likely to miss actions that don't last long enough or fail to capture state changes that occur between two polls. If

the program listens to events, the implementation likely handles only one per button per update tick and ignores the rest.

If your program misses input and you know these two likely causes, you'll quickly track down the source because you only have to look in two places. From there, you can further investigate the reason and possible solutions.

General Software Development

Even if you don't have experience with the particular type of program you're working on, relevant frameworks, or other related technologies, you can still apply your experience from general software development to the problem at hand. For example, if an application freezes, it's highly likely to be related to a deadlock or an infinite loop. If you go through the code to look at all the mutexes, you may very well find the culprit (possibly with the help of a debugger). Alternatively, check all your loops' conditions and make sure they always end. If a program can't write to a file, you know the hard disk may be full or the particular file may be read-only, so you can check that first.

Even if you lack software development experience altogether, you may have some other expertise you can apply. Try treating the program as a black box and observing which inputs cause which issues. These can be anything that influences the program's execution, such as pressing a button, changing a configuration file, or swapping out hardware. Any information about ways to affect the bug lets you make better guesses about its cause.

Focus on Suspicious Systems

Another way to guess where a bug might come from is to investigate suspicious systems. Look for those that have contained multiple bugs before or that are hard to implement without making mistakes.

Check Challenging Systems

Three kinds of systems tend to regularly have bugs across different programs: those that are poorly understood, those that are complex, or those that are used in many different environments.

Poorly understood systems break because the requirements for pre- and post-conditions aren't precise. A programmer is likely to improvise when a

system is underspecified, producing code that may not align with the program's general direction or may have problems with usability or design.

Additionally, poorly understood systems often sit between others. If multiple systems interact, you must understand how all of them behave, as well as the requirements for calling into them. For example, a system that connects multithreaded data processing with the GUI must consider how and when the data processing is ready to accept new inputs and when it can deliver results that the GUI can display. It must also account for processing time, sustained interactivity, and the results of errors or user cancellation.

Very complex systems deal with challenging functionality implementation like network communication, which is hard to get right because it involves a lot of complex pre-conditions for calling each networking function, all results arrive asynchronously, and latency issues arise. If the system sits between a platform's networking API and a program that needs a synchronous interface to get information from some server, these different requirements can cause alignment issues.

Foundational systems also tend to exhibit more bugs than decoupled ones, primarily because people use them more often and in different ways. Consider task schedulers that facilitate multithreading. Most other systems run through them to parallelize computation, handle dependencies between tasks, and prioritize more urgent work. Just one minor mistake can cause serious bugs in higher-level systems.

One clue that a system's implementation was difficult is heavily commented source code or extensive documentation about its internals. If the original programmer thought it necessary to write more about a particular system than the rest of the program, it could signal a higher potential for bugs. Read comments and documentation carefully because they may give you an indication of what aspects were difficult and where you should focus your search.

Check Repeatedly Broken Systems

If multiple bugs have occurred in a system, there will likely be others, which may be very similar because they stem from the same underlying cause or misunderstanding. When facing a new bug, there are better odds that it lies in a system that had many others before.

Fixing previously existing bugs in a system also increases the probability of creating more because you're making changes on top of the original implementation. Unless you're the system's original developer, you won't know every detail about it, its dependencies, and its requirements. This limited understanding may lead you to a Band-Aid solution that will later turn into another bug. Even the original developers may have forgotten a system's intricacies after a long time away. The more a system grows and changes over time, and the more developers touch it, the more likely it will be a breeding ground for bugs. While this isn't true in all cases, a system that shows many problems is likely to become such a hotspot. It would be wise to address this in the long run, but for now, the goal is just to find an entry point for further investigation.

If you've been working on several bugs for a system, you've probably started to build a bit of knowledge about it even if you're not the original developer. Armed with this information, you can more easily attribute bugs to this system and pinpoint them to a particular spot in the code. Additionally, it should be increasingly straightforward for you to fix bugs in this context. A pattern might emerge over time and point you in the direction of the actual underlying problem.

NOTE

There are many reasons why a developer could make a mistake that leads to a bug, including inexperience with the subject matter and deadline pressure. Alternatively, the system may originally have been created for a specific use case and inappropriately repurposed or implemented as a prototype without anyone producing a proper shippable version. Whatever the reason, assigning blame is never helpful; it's better to work constructively and improve the system.

Ignore Irrelevant Systems

In addition to making educated guesses about the likely source of the bug, it's also useful to rule out areas that you don't need to investigate. The bug's behavior should provide hints about what is or isn't relevant. Start with the seemingly relevant systems and only work outward if you don't find an entry point there.

Look for Relevant Responsibilities

Understanding the responsibilities of the systems in your program and framework can help you determine which systems are likely not related to your bug. These usually deal with distinct aspects of the program and don't communicate with the functions most likely involved.

Efficiency in bug investigations is one reason why programmers should strive for separation of concerns and decoupling between the various systems that make up a program. When a system's responsibilities are clearly defined, it's easy to tell what functionality it offers and whether that may be involved with your bug. Additionally, if two systems have clear boundaries and interact only when required, you can more easily follow dependency chains.

Systems that take care of different aspects of your program are usually independent. For example, your GUI system probably doesn't directly communicate with the audio system or the networking layer, and the audio likely doesn't directly interact with the networking. Animations are likewise independent of any databases your program might interact with. Intermediate systems may couple these different systems to drive functionality, but they shouldn't communicate directly.

IN PRACTICE: FOCUS ON THE MOST LIKELY CULPRIT

While working on a larger application, I encountered a problem: Not all command line parameters were parsed correctly. Because the architecture of this program was relatively well constructed and the task of parsing parameters didn't depend on a lot of other functionality, I was able to ignore many irrelevant systems (input handling, GUI, audio, and so on).

I started by investigating how the `main()` function parsed the parameters it received from the operating system, using a small system composed of two files. Everything related to strings, containers, and locales also became part of my scrutiny. For example, I thought the logging system might have had an unintended side effect on the strings it was supposed to log, so I kept it in the back of my mind as I worked.

Ultimately, I discovered that the system responsible for parsing parameters wasn't handling positional parameters correctly. I was happy I hadn't wasted my time examining the program's many unrelated systems.

Of course, one or more of the systems you've deemed irrelevant may turn out to be relevant after all. Some systems are connected in surprising ways, influencing each other thanks to unfortunate design decisions, required dependencies that aren't immediately apparent, or shortcuts developers took when implementing them. All programs are unique and may have special requirements that lead to uncommon and potentially surprising relationships. For example, the audio system may pull sample data directly out of a database, the networking layer may write statistics straight to the GUI, or a 3D renderer may request texture data from files directly.

Consider Surprising Dependencies

Bugs can also create relationships between systems that don't have any intentional connection. For example, memory corruption caused by one system usually affects others, changing data to random values in random locations. This can suddenly link two systems that shouldn't logically have anything to do with each other.

Memory corruption can come from multiple factors, such as race conditions, bad pointers, or external sources like files or devices. When it occurs, your program likely won't behave the same every time you run it. A specific type of corruption in a specific place may trigger one bug, while the same problem in another memory location may create a different bug or may not cause problems at all. (For identifying and removing sources of randomness, see "Remove Randomness" on [page 69](#).)

Another relatively common reason for independent systems to become connected is the use of generic, foundational features. These might include task schedulers, threading primitives, random number generators, or file access functions. Problems with or misuse of these elements may affect multiple other systems, since they may depend on the same low-level functions.

TIP: CHOOSE YOUR RANDOM NUMBER GENERATOR CAREFULLY

If two systems need to generate a random number, each system's sequence should always be independent of the other's. However, if you rely on the good old C function `rand()`, you'll notice that it uses some internal state to generate the next random number, linking any two systems that use this function. Instead, you should choose a

random number generator that allows you to pass in the state. Some libraries provide `rand_r(unsigned int*)` for this purpose. Another alternative is to use C++'s generators from the *random* header file.

Know When to Change Your Strategy

As noted earlier, don't spend too long trying out shortcuts. If they don't quickly guide you to the right spot, try other strategies that may take longer but won't lead you to a dead end. As a rule of thumb, you shouldn't spend more than an hour on shortcuts, and usually you'll need less time because your list of guesses and relevant systems tends to be short. It may take a bit longer if you're dealing with a large program that features many potentially relevant systems.

While shortcuts can reveal some information about your bug, they may not provide details about its root cause or potential fixes. In that case, take your knowledge and move on to other strategies. You should try other approaches after probing with two or three shortcuts, or when you feel your guesses become desperate instead of based on your experience. Remember that you can now cross any system you have investigated with shortcuts off your list of possible culprits.

5. Search the Internet

How? Use your favorite search engine to seek information related to your bug or try your luck on specialized websites that may offer more applicable content. Don't forget to search your organization's intranet for documentation or discussion about your program.

Why? If you're lucky, someone else will have encountered an identical or similar bug before and may have more information than you've found so far. They might even have posted the solution, which you can adapt for your own purposes. Simply finding more about similar issues or other people's approaches may give you hints or new ideas.

IN PRACTICE: FIND SOLUTIONS TO COMPLEX ISSUES QUICKLY

While working on a little tool designed to send network messages to a larger program, I pulled in a tiny library to help send HTTP requests. This library consisted of precisely one header file. However, after including that file and beginning to build my project, I got the following compilation error:

```
error C2011: 'sockaddr': 'struct' type redefinition
```

It looked like a complex problem that would have taken me down a rabbit hole as I followed type definitions in several header files. Before going that route, however, I put the error message into Google and quickly found the most likely cause: You must include *winsock2.h* before *winsock.h* if you're using both. Indeed, just adding an `#include <winsock2.h>` at the top of my code fixed the problem.

Searching online for solutions or possible hints works well when dealing with widely used technology. Programming languages, tools, services, libraries, frameworks, engines, middleware, SDKs, and APIs may have caused others to run into issues like yours. The more users something has, the more likely you'll find relevant results online.

Solutions to the problem you're trying to solve can come in many shapes. You may find information on how to express something in the programming language you're using, a hint about how to get to a specific command or option in a tool, a new binary build of an SDK, or code you can integrate into your own codebase or that of a third-party library.

Internet searches often lead to places that let you ask others for help, like forums, question-and-answer sites, or even social media communities. Do your own research first, but if you can't find any useful information, consider asking others (see "Ask Colleagues for Help" on [page 79](#)). This way, you can get the information you need while contributing to building a shared knowledge base on the internet so that others benefit from questions you've asked or solutions you've posted.

Don't Scroll Too Far

The biggest problem with searching the internet is that it contains a huge amount of information. If you're not dealing with obscure topics or using precise search terms, you could end up with hundreds of millions of hits. Don't waste time sifting through all of them manually.

It's rare to find anything relevant past the second page of search results, so after the first 30 to 50 entries, go back to your terms and options.

Try Different Search Engines

While the landscape of search engines shifts occasionally, Google has been the most widely used general-purpose option for years. Competitors such as Bing and DuckDuckGo serve the same needs, and you can mostly use them interchangeably. Because searching the internet involves minimal effort, it's best to start with the broadest and most generic engine (like Google), then try a relatively generic software development knowledge base such as Stack Overflow. Depending on your bug's topic, even Wikipedia may be useful if you need to look up some formula, algorithm, or other fact. You can then move on to more specialized websites for the general technology you're working with. For example, try the Mozilla Developer Network (<https://developer.mozilla.org>) for anything related to web technologies. If you're working with Microsoft technologies, you may want to try Microsoft Docs (<https://learn.microsoft.com/en-us/docs>).

An answer to your problem could exist on social media, but it may be difficult to find in the overwhelming mass of information. General-purpose search engines often don't index social media data fully or at all, so a dedicated search on social sites may reveal information you can't find otherwise. Try websites like Hacker News (<https://news.ycombinator.com>), Mastodon, Reddit, X, or YouTube. Their community approach may help you discover highly relevant content.

You can also look for even more specialized sites by using a regular search engine for terms like *web development discord* or consulting the official documentation for non-public sources of knowledge. One of your first stops should be the list of frequently asked questions.

Keep in mind that not all websites are publicly available. Normal search engines may not index some of them, and some may require dedicated accounts to access, particularly if they deal with licensed technology or platforms. If you can make it over this threshold, these communities provide valuable content from experienced users and developers.

A single technology will generally have some or all of the following services:

Bug trackers and feedback systems Repositories that list known bugs and their status. In some cases, they include more general feedback and discussions from which you can gather additional information.

Chat servers Spaces for transient impromptu discussion. Some of these, including Discord and Slack, still offer search functionality.

Documentation and knowledge bases Official documentation for the given technology. These services also provide discussion or editing functionality.

Forums Asynchronous spaces where people can read and reply to content threads. This classic form of online discussion can offer a great deal of information.

Public roadmaps An overview of upcoming updates, features, and bug fixes. Roadmaps are more useful if they incorporate discussion and voting.

Support systems Like bug trackers and forums, but with more focus on community-supported solutions. Each resembles a version of Stack Overflow dedicated to a specific technology.

Compared to general searches, specialized services contain fewer seemingly useful answers that later turn out to be irrelevant. The answers themselves are also likely to be of higher quality, allowing you to use them more directly instead of deriving a solution specific to your situation.

Use Advanced Search Engine Features

No matter what service you're using, first try a very straightforward search with the terms you're looking for. For example, you could look for an error message or a crude description such as *C# disable button*. If that gives you irrelevant results or too much information to sort through, start narrowing down your results with the engine's advanced features.

Sorting options alone might get what you're looking for to float to the top of the list. You could sort by relevance (as the algorithm rates it), time of creation, time of the latest comment, and so on. If you're looking for something older, like information about a bug that existed only in previous versions of a library, it's beneficial to tweak these settings. Filters are also

tremendously useful when it comes to reducing the number of results. You may be able to limit your search to a certain section or topic.

Filtering and sorting don't always have convenient user interfaces, so you might need to use operators in the string itself. If you have trouble finding what you're looking for, look into which operators your search engine supports. Here are some useful operators for Google in particular, most of which can also be used in competing engines either verbatim or with little adjustment:

-term Exclude results containing *term*.

"term" Ensure that *term* actually occurs in the results in this form. It also allows forcing the search engine to list only results where multiple terms appear in the correct order, like *"term1 term2"*.

term1 * term2 Match any words or phrases between the two terms.

term1 AND term2 Make sure both terms occur in the results.

term1 OR term2 Allow results that contain one of the two terms.

filetype:pdf Only show results from PDF files.

site:example.com Only show results from the given website.

intitle:term Show results with the given term in the title.

inurl:term Show results with the given term in the URL.

before:YYYY-MM-DD Search for results from before the given date.

after:YYYY-MM-DD Search for results from after the given date.

Be careful to avoid including a space character after operators with a colon or hyphen.

The two most essential operators are *-term* and *"term"*. Excluding a word or forcing the results to include one can do wonders. For example, when searching for information related to the Rust programming language, it's helpful to exclude references to the game of the same name: `rust -game`.

IN PRACTICE: USE QUOTES TO GET MORE CONCRETE RESULTS

I once needed to find a file called `px.cpp` that I knew had been previously available online. Just searching for `px.cpp` didn't get me the right results, but when I entered

`“px.cpp”` into the search bar, the file appeared on the first page. Similarly, when a dialog says, *The error code is 2810.*, searching for *“The error code is 2810.”* yields more relevant results than if you leave out the quotes.

Remember that your code may concatenate strings from several partial strings before showing them to you, which means searching for the whole string won’t yield any results (see “Search in All Files” on [page 86](#)). Any concrete numbers, times, values, or general data from your specific instance of the program will most likely have been inserted into the text before being printed to the log, written to a file, or displayed on the screen. Search for partial strings only and consider using the `*` operator to let any value be present.

Search for Source Code

Searching for source code is tricky no matter what engine you use. Many special characters in code either don’t show up in regular text or structure that text without carrying any real meaning themselves. Because search engines primarily target regular users trying to find text that isn’t code, they ignore those characters. They also don’t generally index popular public code repositories: While it’s easy to find individual users, organizations, and projects on GitHub, you don’t get results for all the code it hosts.

If you really need to find source code, you must go directly to the source, no pun intended. GitHub has a great feature (<https://github.com/search>) that allows searching through all code it hosts with powerful options and filters. While not all online services that host code offer search over all repositories, they typically provide some way to search through specific ones. Find a relevant repository using a regular search engine, then use a specialized query to go through the code. You can also clone an open source repository and search using offline tools if the online options don’t help (see “Search in All Files” on [page 86](#)).

Technologies with generic names make the searching process even harder, especially when they’re new. Here are some examples:

Go Could mean anything even combined with other terms. As a workaround, search for “golang.”

Metal May yield irrelevant results despite the combination. Searching for “metal api” can help.

Rust Can be confused with oxidized metal or the game of the same name. Use “rust lang” instead.

Swift A generic term that also has many other specialized uses (for example, in banking or music). Alternative terms are “swift lang” or “swift apple.”

Don’t Forget Your Intranet

Try looking through any internal wikis, bug trackers, project management tools, and so on. You may even have to query a couple of separate sites, because it’s not common for an organization to have an internal search engine covering all the services making up its intranet.

The results will be much more limited, but any information will be highly relevant to your use case and more specialized than anything you can find elsewhere. If a similar bug has occurred before, the documentation left behind from the previous investigation and fix should be especially beneficial (see “Keep a Debugging Log” on [page 140](#)). Another advantage of finding a source within your organization is that the person who has written down the information may still be in the office, allowing you to ask follow-up questions if necessary (see “Ask Colleagues for Help” on [page 79](#)).

6. Check Version Histories

How? Go through the version control system storing your program’s development history to see if you can find a different version that doesn’t expose your bug. If you do, you’ll likely figure out the significant difference between that iteration and the broken one.

Why? Having identified the significant difference, you can learn what parts of your sources are related to the bug and, potentially, how to fix its root cause.

This strategy is useful primarily for bugs where previously functioning features are now broken.

You can also look into future or parallel timelines. If you're working with a larger team or using third-party software for which you have the source code, other development branches are likely implementing features and fixing bugs and may have relevant changes. Future versions on release branches can also show you updates that haven't been merged into the mainline yet or will never be backported to your version.

Checking other implementations can be helpful beyond the probing stage. If you can track down the change that breaks or fixes the functionality, you'll end up with a pretty good understanding of what is happening and why. With luck, you might even find the correct bug fix.

Find a Working Version

First, you'll need to determine whether the feature in which your bug occurs worked correctly in the past. Your goal is to get a rough idea of when the bug was introduced.

The person who reported the bug may be able to answer this question. Presumably they've tested or used the same functionality before and can tell you when it last worked for them. You can also ask the QA team and other users if they recall a time when the bug didn't exist. If you've used or tested the functionality in the past, you may have your own ideas about the timeline.

In your bug report form, you may want to ask reporters when they last saw a feature working correctly. If it never worked properly, that's also useful to know (see "What Makes a Great Bug Report?" on [page 11](#)). Your QA team often already knows the answer to this question, since they're familiar with the program, but they don't put it in the report.

Go Backward and Forward in Your Version Control System

Once you have an approximate idea of when the bug was introduced, try out a version of your program directly before that point to confirm it. Early on, focus on the latest changes. A freshly reported bug was probably introduced just as recently.

IN PRACTICE: REVERT TEMPORARY CHANGES

Out of the blue, one of my projects received a bug report about overall lousy performance, even though we'd been making performance improvements. I first checked to confirm the issue myself, then profiled the program, which once again confirmed that performance was terrible. However, the profiler couldn't point me to any hotspots or apparent culprits.

How could this have happened? I remembered that the program had been fine two days earlier, giving me a relatively small window of changes that could be at fault, so I went into our version control system and checked every single commit in that time frame.

I quickly found the problem. A colleague had accidentally checked in their changes meant only for local debugging and testing. They had disabled optimizations for several functions, using `__attribute__((optnone))` for Clang, so they could debug them in a release configuration, but of course that meant the functions ran much more slowly. The fix was easy: Just undo those function attributes.

If the version you're trying is already broken, go back further in time. If it's not broken yet, go forward. Take larger steps backward or forward to get started. As you get closer to a point where the bug must have been introduced, narrow down your target with smaller steps. Go back and forth until you've found the exact change that introduces the bug.

If you have a good idea of when the problematic modification happened, this process should be fairly quick. If you have no clue, a binary search is best. Sort the commits to your project's version control system by the time they are submitted, then test whether the one in the middle shows the bug. Continue with the older half if it does, the newer if it doesn't, and repeat until you find the specific commit introducing the problem. Binary search works only when the bug has continuously been present after a certain point. If it gets introduced, fixed, reintroduced, and so on, you won't be able to find the right commit this way.

TIP: USE GIT BISECT

To find a commit using binary search, use the `git bisect` command. This is how it works:

```
git bisect start           # Start the bisect process.
git bisect good commit   # Mark a "good" commit.
git bisect bad commit    # Mark a "bad" commit.
git bisect good            # Mark the current commit as "good"
```

```
d".
git bisect bad          # Mark the current commit as "bad".
git bisect reset # Exit the bisect process and reset the repository.
```

During its execution, this command even tells you how many more commits you'll have to test:

```
Bisecting: 675 revisions left to test after this (roughly 10 steps)
```

While git supports this process with a dedicated command, you can still search manually in git or any other version control system.

Learn from Commit Messages

Finding a relevant change is much easier with meaningful commit messages that summarize what has changed and why, because they may tell you which systems or behaviors are different. Ideally, if the author thought the commit might cause problems, they've added a warning to the message as well. (See "Share What You've Learned" on [page 186](#).) Also check which files were altered in case any of them are related. However, keep in mind that both the commit message and the list of files may be missing important details. Relying on them alone may lead you to ignore a change because it appears irrelevant, only to later discover a connection to your bug.

Decide What to Test

The straightforward approach to evaluating a version of your program is to check out each commit from your version control system and build the code and assets from their sources. You can also use previously built versions if you have them.

Use Existing Binaries

Before building a version for testing, see if you can find existing binary builds you can use instead. You'll have to pick builds for which you can tell

the ID of the commit they were built from, although you can determine an approximate commit if you can tell when the build was created.

You may have older builds lying around locally or be able to access archives from past milestones or the continuous integration system. Other potential locations include storefronts with older versions and clients who received the builds.

Binary builds allow you to easily ask for help from other teammates, especially your QA testers, which is a huge advantage. You can tell others which builds need to be checked, and multiple people can do so simultaneously. Using binary builds also lets you skip building your project, which can be a critical timesaver. Even if you've recently improved your build times, those won't apply to older versions. (To improve your build times for future instances, see "Reduce Build Times" on [page 19](#).)

If you don't have binary builds on hand for every single commit, it's a great idea to combine the ones you do have with building from sources. Narrow down the range with binary builds, then do a more fine-grained search with source builds. You can also check your program's public build for the bug's presence. If the bug doesn't exist there, it must have been introduced since then, which should narrow down the range of commits in your version control system drastically.

Build from Source If Necessary

If no binary builds are available, you'll have to bite the bullet and build from source. Be extra careful selecting which versions you evaluate so you don't waste any time.

Another approach is to mix and match assets and code from different sources. For example, if you know your bug is in the assets and not the source code, you can keep the same code and try different versions of the data. If your program allows you to combine binaries from an existing build with pieces you've built locally, you can reduce iteration time even further.

Swapping out just one small piece every time you build and test makes your iterations much faster. If you've already identified a single asset, source file, or function in the code as problematic, try different versions of just that feature. It might take some fiddling with your version control system to extract only one file or a part of a file from a specific version out of your repository, but this can ultimately save you lots of time.

Identify the Offending Change

Once you've found the commit that introduces the bug, analyze it to find the problematic difference in the new code. While it's theoretically best practice to create commits that contain only one change, this is usually not the case in reality. Unrelated changes are often committed together. Moreover, even a single change typically consists of multiple modifications to multiple files, including both source code and assets.

Review the modifications in the commit one by one to find which causes your bug to occur. For larger commits, you may want to employ a systematic approach once more, maybe even a binary search.

When you locate the change you need, start investigating how to undo just its problematic component. Remember that the commit's author had reasons for building it the way they did. The goal is to find a way to undo the problematic change without undoing the actual improvements.

To that end, try to cooperate closely with the author if they're still available. They should be able to provide extensive background information and may even have theories as to why the commit introduced a bug; we've all made changes that felt a bit risky or that had an unclear impact.

The best way to revert or tweak the problematic change highly depends on what was changed and how. It may be easy to see how a modification to the source code introduced a bug, but in binary asset files, it's often less obvious. You may even consider handing the bug over to the commit's author, depending on your respective skills and workloads.

Look into the Future

Instead of just looking at the program's past, it may also be worthwhile to peer into its future—that is, examine commits that come chronologically after your current project state. Someone else may have already knowingly or accidentally fixed the bug.

While it's possible for this to happen with smaller teams, the likelihood increases with team size, often due to bad planning or miscommunication. A one-person-per-bug policy avoids these problems, but if a team is big or multiple teams work on a program, it becomes harder to follow.

Looking into the future is likely to be most useful for programs that integrate third-party products. Bugs in such products have likely affected

other people and may already have been fixed.

Finding Bug Fixes for Third-Party Products

If you find a bug in a third-party product, check whether it has already been fixed. If any new versions are available, read the accompanying changelogs. If they don't provide any useful information, the third party might be able to supply a roadmap or a public bug tracker if you ask.

If you have access to the product's source code, you don't have to rely on changelogs or secondhand information but can directly confirm whether your bug has been addressed. The version control system will also let you analyze individual commits for any work related to your bug. Changes made after your version was released could be particularly helpful.

TIP: ALWAYS PUSH FOR SOURCE CODE LICENSES

To reduce the risk that comes with licensing and integrating third-party libraries or frameworks, always try to acquire a source code license. If you don't, you'll likely end up running into a bug in the closed-source blob without being able to fix it yourself. You'll have to wait for the third party to address the problem, and your bug might not be their highest priority. If this happens at a bad time, like when you're about to release your program, it will have a significant impact on your ability to launch. In the worst case, it can jeopardize your whole product.

With source code readily available, you have the power to get your hands dirty and fix the bug. You can analyze what's happening and come up with workarounds or otherwise make up for whatever has gone wrong in the library. Often your bug will turn out to be a minor issue in the third-party code triggered through some special case usage, and that's easily fixed when you can reference both your code and theirs.

Additionally, having the source code reduces the impact in cases when the third-party developer no longer offers support. If they become unresponsive or go out of business, you have a backup solution and can take matters into your own hands.

Integrate an Existing Fix

If a fix to your bug already exists in a newer version of the third-party software, the next step is to integrate that fix into your program. This process ranges from straightforward to very complicated, depending on how you approach it and how many changes you need to incorporate.

Let's say you've integrated the third-party product as a binary package and it's only a bug-fix release. Just update the dependency and you're done.

On the other hand, if you've integrated the third-party product in the form of source code, you must be careful not to diverge too much. If your team has made several far-reaching modifications to the third-party code and the creator does the same with its version, the update gets tricky. You'll have to merge the changes from both sides while adjusting your own program to deal with any changed behavior in the product. Integrating all these changes can take weeks if the source code is big enough,

Most middleware developers offer source code licenses, and with services like GitHub, access to the development repositories themselves is more widespread. Open source projects almost always make both available (presumably a factor in their popularity).

Having access to the source code gives you the choice between upgrading to the release that contains your bug fix or backporting that fix into the version that you have. Depending on the fix's complexity, applying it to your software is often much simpler.

If it's not easy to backport, and a new release containing the bug fix hasn't been released yet, knowing when it will come out is valuable information as well. It gives you a perspective and tentative timeline, and might also motivate you to update earlier.

IN PRACTICE: USE FIXES BEFORE THEY'RE RELEASED

In my work with games, I rely on the massive Unreal Engine, one example of a large third-party project. The developer shares the source code with anyone using the engine. During the production of a game, we update to new versions regularly until we come to a point when a full upgrade becomes too risky.

Unreal Engine is under active development, and so is our game. Because an update takes its fair share of time, we schedule them as tasks in our product backlog and pull them into sprints when needed. Sometimes we discover a bug in the engine between scheduled updates and urgently need to address it. In such cases, we look for a fix in Unreal's version control system and bring it back into our copy if we find it. Doing so is often possible even when Unreal hasn't released an update that includes the fix.

Look into Branches and Forks

You can follow one more path to uncover possibly helpful iterations of your software: alternate versions or development branches. Software development commonly involves branches for specific tasks, fixes, hotfixes, or releases. Such branches can contain solutions for the bug you're working on, but they may not yet have been merged into the mainline.

Understanding a repository's branching strategy makes it easier to find the right branch. One common model is Gitflow (<https://nvie.com/posts/a-successful-git-branching-model/>), a trunk-based strategy that promotes dedicated branches to individual features or bug fixes.

Always keep an eye out for *forks*, complete but altered copies of a project that someone other than the original creator usually develops and maintains, which commonly exist in open source projects. Some take directions that may align more than the original with your needs, while others aim to fix issues for use cases that may fit yours perfectly.

If you find a relevant change in a fork or branch, you have a difficult decision to make. It may make sense to completely switch to that fork or branch to integrate the third-party product. The alternative is to apply and personally maintain just the one change you were looking for.

Switching means you won't be getting any updates from the original developer, which may still be important to you. On the other hand, just integrating the one relevant change means you'll have to make sure it works with any future versions.

7. Attach the Debugger by Default

How? Whenever you launch your program, launch it through the debugger.

Why? A debugger allows you to inspect your program's behavior and data. If your software runs into any bugs during execution, you can immediately investigate the cause. The program may report bugs itself using an exception mechanism, or you can place breakpoints.

Running programs with the debugger attached should be standard procedure for every developer. Debuggers let you observe how the computer executes the template for the logic you've provided in the form of source code. If

you don't leverage this tool to kickstart your investigation, you'll waste time by retracing steps the debugger would have quickly presented.

Inspect Your Program's Behavior

A debugger's core features let you inspect your program's inner workings and see the values of any variables. Debuggers also allow you to step through the control flow and see any decision as the computer makes it, then use breakpoints to pause execution when the computer reaches a certain point in your code. While you can add many more advanced features, this minimal set already makes for a powerful tool.

In a program, your code serves as a template. The execution fills it with values for your variables and reaches decisions for all your control-flow instructions. Unfortunately, there's often a disconnect between template and execution. When reading the code, you might assume that a variable has a particular value or the software will follow a specific branch of your `if` statement, but neither assumption may hold true in reality. With the help of a debugger, watching this discrepancy in real time makes finding the cause of a bug a lot more efficient.

For example, it's easy to determine how many times this simple loop executes:

```
for (int i = 0; i < 10; ++i) {  
    printf("%d\n", i);  
}
```

However, if we add more complexity, it may not be possible to guarantee the number of loops. Suppose a global variable is part of the loop condition. Suddenly, any other thread or function call from within the loop can influence how often the body will run:

```
int global_var = 10;  
void print_number(int i) {  
    global_var = printf("%d\n", i);  
}  
void f() {  
    for (int i = 0; i < global_var; ++i) {  
        print_number(i);  
    }  
}
```

```
}  
}
```

Similarly, loops can be interrupted with control-flow statements that use a `break` to jump out and `continue` to start the next iteration. The same goes for a `return` statement that immediately interrupts the loop and jumps out of the current function. Other control-flow statements, such as throwing an exception or using the infamous `goto`, can trigger exits from deep within the call stack without notifying you. Execution will resume far outside the loop you were looking at.

Understanding such behavior just by looking at the code is difficult. However, the debugger may take you to a completely different place in your source code than you expected, making it easy to comprehend the jump and follow the logic.

Use Exception Handling

The debugger can automatically break execution if certain types of errors occur. For instance, if the program crashes, the debugger will direct you to the part of the code that triggered the crash. The call stack shows you where the call to this crashing function came from, and the debugger will reveal any variables it can salvage.

You may be able to break your program's execution when it triggers a certain type of exception, depending on the programming language you're using. (This technique is generally more available for languages that involve management or scripting.) For example, you can configure your debugger to break execution whenever your C# code throws a `System.AccessViolationException`. You can easily find the source of the exception without manually identifying all potential sources.

If you're working in C or C++, you can rely on the structured exception handling (SEH) extension, which Microsoft supports in its compiler and runtime. SEH enables you to catch hardware faults and other issues that standard language features can't detect, such as access violations or division by zero in native programs. It comes in handy if you want to detect such a problem and investigate it further, but in most cases, you shouldn't try it in a shipped product or even try to recover programs using it. It's a debugging mechanism and not meant for error handling.

The Visual Studio debugger may automatically break when it encounters exceptions. A program's state can be challenging to analyze after an exception has been raised because the call stack or data may be broken due to corruption, but the debugger can often show you where this kind of problem originates.

Use Breakpoints to Inspect Your Code's Execution

Most bugs don't automatically prompt the debugger to indicate the line of code that caused problematic behavior. Instead, you must break the execution on any lines of the source code that you suspect are related.

Start by putting breakpoints in functions within potentially problematic features or systems or functions that call into them. Position the first ones at the top of a function or even higher in the call stack; the execution may not hit them otherwise due to early returns or other control-flow conditions. Place breakpoints outside of loops and conditional statements even if you assume that the loop or the body of the `if` statement will be executed. After all, the fact that you're debugging proves you can never be sure that all your assumptions are accurate.

Limit your first set of breakpoints to a few functions deep in the stack, closest to where you hypothesize your bug lies. Triggering too many generic breakpoints (that is, in functions high up in the call stack) can make it harder to determine which ones are actually related to the bug.

If the program doesn't trigger any of your breakpoints, go higher in the call stack and place increasingly generic breakpoints until you've found at least one that reproduction hits. From there, debug downward into the functionality to find the point where the bug occurs. Use your debugger's *step into* and *step out* commands to drill into call stacks. In the worst case, you'll have to start this process from the top of your program, traditionally the `main()` function. Step through the code and into function calls from a promising breakpoint to find exactly what triggers your bug (see "Step Through the Code" on [page 100](#)).

For native code, breakpoints are cheap, and you can make liberal use of them as long as you stick to the call stack depth you're currently investigating. They're only a keypress or mouse-click away, and they don't affect your runtime performance because they're implemented in hardware. However, be sure to remove all associated breakpoints once you've finished

working on a given bug, as going back to find those you’ve previously set can be frustrating.

Your IDE likely has a way to delete all breakpoints. If you want to retain some (for example, because you expect you’ll have to work on this bug again), most IDEs allow you to disable them and reactivate them later.

NOTE

Asserts can also trigger breakpoints. However, they aren’t set through your debugger but added to the code. Since this requires a recompile, using the debugger’s breakpoint functionality usually makes much more sense.

Set Up Debuggers on Difficult Platforms

For some platforms, debugging is extremely cumbersome or isn’t supported at all. Even in these cases, it’s usually worth getting an appropriate tool up and running. If you develop your program without debugging any of your script code, fixing any bugs that arise will be time-consuming and frustrating, especially as you’re putting on the final touches before release.

If your platform doesn’t include a built-in debugger, investigate third-party solutions. Ask the platform provider for help and search through community resources. Alternatively, you could write your own debugger, which is realistic only if you’re working on a long-term project that heavily relies on the debugger-less platform. A custom-built debugger takes a lot of time, but it also integrates more seamlessly. As such an endeavor is very specific to the language and platform you’re using, you’ll have to follow the relevant documentation closely.

When you can’t integrate debugging functionality yourself, consider reporting that to the platform owner. They might not know that debugging doesn’t work under certain circumstances or how important a quality debugger is to developers. Submit the issues you encounter as bug reports so that the development tools team can efficiently respond and fix the bugs you’ve encountered (see “What Makes a Great Bug Report?” on [page 11](#)).

Handle Cases Where the Debugger Interferes

In rare cases, it’s excusable not to use a debugger. The debugger may interfere with execution, such as by preventing race conditions or low-level

exceptions, or the program might detect the debugger's presence and run differently based on its own logic.

One possible workaround for such situations is to attach the debugger after you've reproduced the issue. You may see the bug's results in the values of variables and the control-flow decisions. From there, you can backtrack manually and determine how the code ended up in this failure condition.

The debugger may also cause problems. It's possible, for example, that your program crashes *because* the debugger is attached. Instead of working around this situation, investigate why this happens and try to fix the problem before you continue with the bug you originally intended to work on, as ignoring it can severely affect your productivity. Sometimes, however, the problematic debugger behavior is so intertwined with the original bug that you'll have to remove the debugger and resort to one of the other strategies in this part of the book. It may make sense to do so if you're short on time, such as when you must fix just one bug before an urgent release.

8. Instrument the Program to Report Problems

How? Modify your program with instrumentation to report issues or make them more evident.

Why? These changes allow you to more efficiently spot irregular behavior that might be the source of your bug. Your program can help you with that because it has access to all the information surrounding its execution. It may be able to tell you what the bug's root cause is, where it's located, where the symptoms manifest, and why it happens. While in most cases you won't be able to determine the fix just by reading what the program provides, it gives you a good starting point for deeper investigations.

Though it may seem counterintuitive, this strategy involves modifying the code before fixing the bug or knowing any details about it. You must be very careful to avoid making changes that influence the bug and its reproducibility.

This section covers how to add assertions, handle errors returned by functions you're calling into, and initialize any data related to your bug with patterns that are easy to spot. As some of these techniques are common programming practices, they may already appear in your program, but there's likely room for improvement in the context of your particular bug.

Use Assertions

Broadly speaking, an *assertion* is a check that determines whether a given condition is true. If not, it prevents the program from continuing, providing an easy way to express conditions for the code following it and avoid follow-on issues.

As you write code, you make many assumptions about *pre-conditions* and *post-conditions*. Liberally add assertions to express these assumptions and verify them. They'll make your life much easier by allowing you to detect cases where your assumptions aren't true.

You can use assertions for pre-conditions to ensure your assumptions are true before executing the code that relies on them:

```
float divide(float x, float y) {  
    assert(y != 0.0f);  
    return x / y;  
}
```

Or post-conditions to enforce the result of an operation to be what you expect:

```
std::vector<int> ints;  
ints.push_back(0);  
assert(ints.size() == 1);
```

Or, more generally, any assumptions you're making:

```
double calculate_hypotenuse(double a, double b) {  
    double squared_sum = a * a + b * b;  
    assert(squared_sum >= 0.0);  
--snip--
```

Most programming languages support asserts in one form or another. C requires the *assert.h* header file and then offers the `assert()` macro. For C++, it's the *cassert* header file and the `assert()` macro as well. Python also has `assert` statements. In C#, you need to `import System.Diagnostics` and then get access to the `Debug.Assert()` method.

Assertions don't free you from checking for errors and results. It can be difficult to decide whether you should write reliable, defensive code or add an assertion. When it comes to a likely common issue, the code should do everything it can to avoid erroneous behavior. However, because implementing those checks may take too much time or present too much of an implementation overhead, you may opt to add an assertion instead, especially if it's only for the rare case of a potentially broken pre-condition.

In many cases, including both an assertion and proper error handling makes sense. An assertion should alert you when running the program through the debugger or testing. In a shipped build, on the other hand, you don't want your program to simply give up and crash but to handle the error gracefully and continue execution.

TIP: DISTINGUISH BETWEEN ASSERTS AND ERRORS THAT CAN BE HANDLED

You should use asserts for exceptional circumstances and error handling for situations that can occur during regular program execution. The following code shows an example.

```
int write_to_file(const char* filename, const char* content)
{
    assert(filename != NULL);
    FILE* file = fopen(filename, "w");
    if (file == NULL) {
        return -1;
    }
    --snip--
}
```

An assertion checks for the unlikely event of the caller passing in a `NULL` pointer for the filename. In contrast, writing the check for whether the file was successfully opened as a regular `if` statement allows the function's caller to handle this error and react to it appropriately.

Always place your assertion as early in the code as possible. The goal is to make your program fail wherever your assumptions are first violated, which will help you understand where the issue comes from and, ultimately, how to fix it.

Get Information from Assertions

When an assertion makes your program stop, the program shows an error message that tells you which condition failed. Depending on the language and implementation, the message may include additional information, such as the filename, line number, and potentially even the values that produced the problem. If possible, you should add a custom string to the assertion that allows you to include additional context.

Usually, an assertion exits the program without allowing further analysis. When running inside a debugger, however, an ideal assertion implementation will cause the debugger to pause execution as if a breakpoint had been triggered. This is tremendously useful because it allows you to start debugging immediately. You may need to tweak the assertion or implement your own to allow such behavior.

A custom implementation that triggers a breakpoint on Windows can be very straightforward:

```
#define ASSERT(cond) if (!(cond)) {__debugbreak();}
```

You may want to add a string:

```
#define ASSERT(cond, message) if (!(cond)) \
{printf("Assertion failed! %s\n", message); __debugbreak();}
```

It's possible to continue execution after an implementation that triggers a breakpoint, but you should consider the severity of violating the given condition. While you should naturally stop if a catastrophe will result, you may learn more about the code if you're able to let the program run. You may find that the assertion is the problem or use the insights from debugging and being able to continue the execution to improve the error handling in your code.

Use Static Assertions

Regular assertions are executed at runtime and check conditions dynamically based on what's happening. However, static assertions are checked at compile time and help ensure that the program fulfills those conditions. For example, you may want to check that an `int` is 4 bytes in size:

```
static_assert(sizeof(int) == 4, "int has the wrong size!");
```

Compilation will fail if the size of an `int` should be anything other than 4.

Only languages that make use of ahead-of-time compilation, such as C++11 (`static_assert()`), C11 (`_Static_assert()` and `static_assert()`), Rust, and D, support static assertions.

Add Log Output

Like assertions, text your program produces can grant insights into what's happening during execution and where problems arise. In particular, it shows how the program behaves over time and how its data changes, which may allow you to see where something started going wrong (see “Debug with `printf`” on [page 158](#)).

Handle Errors

While assertions allow you to handle potential issues without worrying about how to recover gracefully, a cleaner alternative is properly addressing errors and warnings as they come up.

Always fix the errors that functions you call into raise. If one returns a value indicating whether it was successful, always check that value, irrespective of whether it stems from your code, other people's code, or even the standard library. For a success value, continue as if nothing had happened. For an error value, report the error and try to recover from it.

It's tempting to leave out error handling when a program first comes together in a prototype state. You can simply run the program from a debugger to inspect potential errors, so you really don't need to have the code report problems or even recover from them. This is acceptable in

development versions, but you have to deal with issues before releasing software. Who you report errors to depends on what kind you're dealing with. You should log errors that the program can recover from by itself, but inform users about errors due to bad input or those where recovery requires outside support.

IN PRACTICE: ALWAYS CHECK RETURN VALUES

I once worked on a tool designed to set up the development environment for newcomers to a project. It performed a range of checks before creating the environment to ensure all the required prerequisites would be satisfied and that the installation, repository cloning, and necessary configuration would finish successfully.

This worked well in most cases, but a problem arose: Windows doesn't allow you to install new software if there's a reboot pending. The solution was to add checks between almost every step to ascertain whether that was the case. If so, there was nothing we could do but tell the user to reboot their machine as soon as possible and continue the setup afterward.

It turned out that some installers will tell you they require a reboot after finishing, using an exit code of 3010. Checking the return values of any finished installer executables for this code made the pending reboot detection a lot simpler.

Recovering from errors when possible is essential to providing a stable program that can deal with a wide range of environments. Methods for doing so depend on the specific error. You may have to:

- Call a function again with a different parameter. For example, creating a temporary file with a random name may fail if the name already exists, so you'll have to generate a new one and try again.
- Call a function again at a different time, such as in the case of a temporary network error.
- Call a different function, such as when a video container holds data in a different format than the original function would decode.
- Skip some functionality altogether. If you can't initialize a touch device, for instance, its input won't be available.

If the solution to recovery affects the user experience, be sure to communicate this. For example, if an action takes longer because the program has to retry it several times, notify the user that they'll have to wait

a bit. You should also communicate any missing functions, whether that means a grayed-out menu entry or a modal message box.

Signal Errors to the Caller

Signaling errors is very important but often doesn't have a great solution. While some programming languages have a single standard method, any given one generally offers multiple competing approaches.

Error codes, numbers that a function returns to say that something went wrong, are a simple and common approach to signaling errors. Enums work similarly but provide a better overview of what error types to expect. More sophisticated techniques include dedicated result types and optional types, which can return data in the case of a successful execution or more extensive data when an error occurs. Expressing an error with numbers is still convenient in those cases because you can easily check for them in code, but text can give you more context or provide the user with detailed information.

Many programming languages also support exceptions. These, much like assertions, can bring the whole program down unless you handle them. While that can be a positive, exceptions can also complicate the control flow.

Use Obvious Test Patterns

You can also fill your data with recognizable, obvious patterns, often known as *magic numbers*, *magic values*, or *sentinels*. Since you'll recognize your magic values anywhere, if you set your data to one under particular circumstances, you can easily determine whether specific events have taken place. (While you could also explicitly track status separately, remember that this is about situations where not everything goes according to plan.)

The advantage of filling all your data with a pattern of magic values is that you can spot the pattern in any section. That shows you whether the data has been modified, meaning that a given section of code has run, and which parts have been altered. The last is especially useful in cases where data gets written accidentally.

As an alternative, set up guard values or fences that signal the beginning or end of a memory section. They identify that section and help

you spot whether code has written past a segment it wasn't supposed to exceed.

A magic value should be something obvious. One approach is to null all newly allocated memory by setting it to `0x00`. The operating system itself often does this to prevent data from leaking from one process to another. However, this value isn't evident, so it may not fulfill the intended purpose and may even hide bugs. Magic values should be rare in the data you usually encounter. Commonly used values are `0xdeadbeef` ("dead beef") or `0xbaadf00d` ("bad food"), for example. The best magic values have some special properties (from Steve Maguire's *Writing Solid Code* [Microsoft Press, 1993]):

- They should be odd numbers, as that increases the chance of causing an error when interpreting them as memory addresses.
- Code that operates on them should do something unusual.
- They should cause an error when interpreted as code.

TIP: WATCH OUT FOR VISUAL STUDIO'S MAGIC VALUES

The Microsoft Visual Studio C runtime makes liberal use of magic values in debug builds. Any newly allocated memory is filled with repeating `0xcd` values to signal uninitialized memory. If you stumble upon this pattern in one of your variables, you know you haven't written any sensible data to it yet.

Any deallocated memory is filled with `0xdd`. If you run into a range of memory containing this pattern, you can be sure you shouldn't be reading or writing it. Visual Studio also fills buffers on either side of your allocation with `0xfd`, so you should know never to read or write there.

Magic numbers are often expressed in hexadecimal because it's a compact representation that people can easily communicate and memorize. Additionally, magic numbers often have bit patterns that become visible in hexadecimal. For example, `221` doesn't draw much attention, but `0xdd` raises awareness.

When looking for magic numbers in your debugger, configure it to show variable values as hexadecimal and print the data that might contain them the same way when you're logging it. You can also observe the

memory surrounding a given variable or block, using the debugger's memory viewer, to determine whether you're near an area marked by magic values.

Don't Break the Reproduction with Modifications

The approaches described in this section all require you to change the code to learn more about your bug. This is always a risky endeavor because it can have unintended side effects. For example, adding an assertion or logging output may make your bug disappear or trigger a different bug, preventing you from investigating the one you initially set out to fix.

IN PRACTICE: FIX A BUG THROUGH LOG STATEMENTS

I recently joined a team that had just shipped a very successful software product. They told me that in the time leading up to the launch, they'd found a bug that, for the life of them, they couldn't figure out. It didn't occur when they used a debugger, so they tried adding log statements. Sure enough, that also made the bug vanish. They ran out of time to find a more thorough solution and ended up shipping their project with those statements to fix the bug.

Though no one ever tracked down the bug's root cause, it's possible to deduce what was potentially behind it. Adding log statements changes the timing of the executed code, which can avoid bugs caused by a particular sequence of events, often in connection with multithreading. Alternatively, the log statement may have led to some code or data ending up in a different memory location, avoiding corruption.

While this solution worked in this case by pure luck, you should never enact it on purpose. Any other modification might undo the perceived fix, and you may have to deal with the bug again at an even worse time. Always try to understand a bug's root cause and resolve it properly.

Be very careful when simultaneously debugging and modifying the code to make sure your changes don't affect the bug. Avoid altering the code more than necessary and confirm that you can reproduce your bug after each modification.

Make the Instrumentation Optional

After implementing any of the ideas covered in this section and resolving your bug, you'll have to decide what to do with the changes you've made. You might have the impulse to delete the added code because it doesn't

seem like you'll ever need it again. However, bugs have a way of coming back, and different bugs may appear in the same area of code. In these cases, the debugging support code may still prove helpful.

Error handling is especially useful to keep, as it makes the program more robust and gives the user better feedback, meaning they may be able to fix problems related to their input or environment by themselves.

To keep your support code around, make it possible to enable and disable your assertions, test patterns, and log outputs based on whether you need them. You need to turn them off by default for any builds you send to the user. Enabled debugging functionality tends to hinder performance and may show cryptic messages or cause effects on behavior that seem like bugs themselves.

For shipped builds, use a compile switch that defaults to off for any debugging functionality. You can then enable individual features for certain systems or areas of the code when you return to debugging them. A flag to turn on all instrumentations can also provide an overview of everything your program reports, which might help with a future bug.

9. Remove Randomness

How? Look for functionality affected by randomness, then disable the source.

Why? You can improve your bug's reproduction frequency by disabling randomness. If the randomness is unintended, following it to its source is a great way to find a bug's root cause.

Computers are deterministic, which implies that bugs can't be random even when they appear to be. A "random" bug has a clear cause in virtually all situations. Investigating the possible sources can help you track down the culprit and remove the random behavior.

Create Deterministic Programs

Computers are designed to behave deterministically, so getting them to generate randomness in the first place is hard. If a bug appears to be random, it must depend on something that provides random values or

emulates randomness. Usual suspects include pseudo-random number generators or features that rely on the current time.

You should generally aim to create fully deterministic programs. Besides debugging, this helps in many use cases, such as:

Testing If software always behaves the same way, testing it is easier. Deviations are also much more noticeable and quickly detected.

Automation Implementing any kind of automation is more straightforward when you can make certain assumptions about the results.

User experience A program that reliably provides the same results from identical user inputs is simpler to navigate and generally leads to a better interaction.

Robustness If your code can depend on other parts of the program always doing precisely what they promise, the whole product will be more reliable.

Networking If two or more programs connected via a network can perform the same calculations and come to the same results, they have less need to communicate.

Despite these advantages, it's difficult to achieve determinism in most cases. The whole program must play along, and the more complex and extensive it is, the harder it will be.

Implement Complete Determinism from the Start

If you can confirm that your software needs complete determinism and you can implement it, you have to build it into your design and code from the start. Different compilers for different platforms must generate compatible code. Any third-party libraries and frameworks you've integrated must also be deterministic. It's a lot to ask. Libraries used for networking or complex calculations, such as simulating physics, often lack determinism or are deterministic only under very specific pre-conditions.

IN PRACTICE: COMPLEXITY MAKES DETERMINISM HARDER

My colleagues and I once looked into making a game completely deterministic across different platforms and hardware targets. However, the more we investigated, the more

daunting the task became.

For a fully deterministic game, all systems have to work together. Not even one can drop the ball. All the game's pieces must also be deterministic so that all seemingly random variations are intentional. The simulation of the world must take the same steps every time. When the player presses the same sequence of buttons, the player character must perform the same actions every time. The same nonplayer characters must spawn in the same position, and their AI must act and react in the same way the player does. And so on.

Floating-point math is also tricky. In practice, regular floating-point math is deterministic, but only if the operations are done in the same order. Due to their finite precision representation of real numbers, not all operations yield mathematically correct results. As a simple example, $1.0f/5.6f$ doesn't give the same result as $0.01f/0.056f$ does. These errors can accumulate and lead to slight variations in results when calculations are performed in different orders or on different hardware architectures.

You'll probably run into issues before long, especially if you're using your compiler's fast-math option to improve performance. The option lets any math operations take shortcuts that vary from platform to platform and compiler to compiler instead of generating code compliant with the IEEE standard for floating-point math.

Your operating system can also get in the way of perfectly deterministic execution. Its scheduler decides when to run which program and threads, on which CPU core, and for how long. If your program doesn't always run together with the same set of other programs, or the scheduler itself isn't deterministic, it's harder for your multithreaded code to run in a reliably deterministic way. Even if your code isn't multithreaded, the scheduler influences its execution time by letting other programs run intermittently, which may in turn influence your software's behavior.

For example, if your simulation reacts to variable update rates using a variable time step, you'll most likely rely on a floating-point delta time variable. If the update rate varies, delta times fluctuate, which leads to different results because of the floating-point math's precision issues.

You should generally use a fixed time step when implementing a deterministic simulation. This way, the simulation runs exactly the same way in every update. If the program is well within the budget for the update

rate you want on all your target hardware, implementing a fixed time step doesn't require further effort. However, if your program takes longer to simulate than the time available for one update, it may run slower than intended and feel sluggish to the user. You must add a different mitigation to catch up with the correct time, like running multiple simulation steps per update with a fixed time but not showing all of them. This may affect performance even further, but aids with reproducible simulation results.

Implement Determinism Where It Matters

While you would ideally make your entire program deterministic, luckily you don't have to because it's just as effective to make only the feature or system in which your bug occurs as close as possible. Once you've located an area related to the bug and you've determined that it has a random element, look for the sources and disable them.

Disable Potential Sources of Randomness

Start by paying close attention to the circumstances in which the bug appears. Is it always after a set number of minutes? Does it occur when someone happens to press three keys simultaneously? Does it happen repeatedly or just once? These occurrences create a pattern from which you can learn valuable information.

Once you've found a source of randomness, look into removing or disabling it. This step is a lot easier if you generally have an abstraction covering the sources in question. While that also makes it more difficult to trace those dependencies, it allows you to hook in and override any such sources with your own implementation. You can force them to use specific values or implement your own generator so you get a deterministic sequence.

Time

One possible source of randomness is timing. This could come from the current real-world date and time. However, it could also be related to the time elapsed since the beginning of the program or in relation to other events. The less predictable the timing, the more random the resulting behavior will seem.

For example, in this code, every now and then an out-of-bounds access to the array will occur:

```
int array[10];  
array[time(0) % 11] = 0;
```

Finding spots in the code where the current time is retrieved is relatively straightforward. Depending on your programming language, its standard library, and other frameworks you're using, you can search for any relevant function names. Tracking down a source of time unrelated to the wall clock is more complicated. If your program counts the time itself, you may want to check whether it's accumulating a delta time that is, in turn, based on the wall clock.

Some sources of randomness are easier to find than others. Searching for specific places is tricky when you don't know what something is called. While many functionalities have well-established standard names, others may have common names or various names depending on the platform, language, libraries, and so on. Make sure to search in a smart fashion (see "Search in All Files" on [page 86](#)).

TIP: CONSIDER THESE TIME-RELATED FUNCTIONS

C++ has functions such as `std::time()`, `std::chrono::system_clock::now()`, and `std::chrono::steady_clock::now()`. In C on Linux, you might look for functions like `time()`, `gettimeofday()`, and `clock_gettime()`. On Windows, you have `GetSystemTime()`, `GetLocalTime()`, `GetTickCount()`, and `QueryPerformanceCounter()`. In C#, you can get the time using functions such as `DateTime.Now`, `DateTime.UtcNow`, and `DateTime.Today`.

Python allows you to retrieve the time using functions and modules such as `time()` and `datetime.datetime.now()`, and the `time` module for various time-related operations.

You may be able to make time deterministic by always setting it to a specific value. If that would impede progress, always start from the same point in time and then let the program take its course from there. For delta times, always choose a fixed value such as 16.6ms or 33.3ms, which correspond to a refresh rate of 60 and 30 frames per second, respectively.

Random Data

Randomness may also occur when software reads from an area of memory that doesn't hold any sensible data. Data may be corrupt because you haven't written to it yet or the variable hasn't been initialized properly. While the operating system and programming language will dictate a variable's contents, it'll often contain garbage leftover data from some other program that ran previously. The following example prints a random value, showing there is random data at the memory location `data` points to:

```
int x;
printf("%d\n", x);
int* data = malloc(10);
printf("%d\n", data[0]);
```

Alternatively, you may have a pointer referencing the wrong data. If that gives you seemingly random results, you're lucky, as reading or writing an arbitrary memory location will likely cause a proper crash as the operating system tries to protect itself and other programs. This example shows a crash whenever you try to access `ptr`:

```
int* ptr = (int*)0x80ddff00;
*ptr = 14;
```

Finding improperly initialized variables and misdirected pointers is tricky. With a small scope, you read line by line and make sure all variables have reasonable values, which is straightforward in a debugger. Otherwise, try to reduce the scope. You can also use external tools or let the compiler help you (see “Use Off-the-Shelf Tools” on [page 164](#)).

To ensure all your variables, pointers, and memory blocks are initialized, first research which types of variables your language initializes by default; you can let those be. Also check whether your runtime can help you with this, as discussed earlier in “Instrument the Program to Report Problems” on [page 62](#). For many variables, the rest is simply busywork. Line by line, review all code in any way related to your bug and make sure all variables are correctly initialized to default values.

You'll mostly want to set the default value to zero, a process that involves filling blocks of memory with zeros. Again, an abstraction helps here because you must follow each function that allocates memory with a call to a function that fills all that memory, like this:

```
void* my_malloc(size_t size) {
    void* ptr = malloc(size);
    if (ptr) {
        memset(ptr, 0, size);
    }
    return ptr;
}
```

With this kind of abstraction, you can easily enable and disable your allocated memory initialization. You can also easily debug and track all allocations because they go through one function.

Concurrency

Most platforms support multitasking, meaning multiple programs can run simultaneously. They share system resources, so your software will behave differently between multiple executions because it can use different resources each time. The operating system's scheduler may do the rest and let your software and its threads run in nondeterministic ways.

Additionally, most devices nowadays have multiple cores, which run several programs and threads truly in parallel. If you're not careful, you can quickly end up with a program that corrupts its own data.

The following example shows a sequence that will have different results for `counter` every time it runs:

```
int counter = 0;
void increment() {
    for (int i = 0; i < 100000; ++i) {
        counter++;
    }
}
int main() {
    std::thread thread1(increment);
```

```
std::thread thread2(increment);  
thread1.join();  
thread2.join();  
--snip--
```

Finding entry points for multithreading is simple. Again, depending on the programming language, standard library, and framework, a couple of relevant functions and types can facilitate starting up new threads and managing multithreading primitives such as mutexes, semaphores, and events. However, it's harder to determine what logic runs in parallel. You can investigate this in the debugger by modifying the code to display what logic is currently running or by excluding concurrency using mutexes (see “Step Through the Code” on [page 100](#)).

TIP: LOOK OUT FOR THREADING-RELATED FUNCTIONS

In C++, you have `std::thread`, `std::mutex`, `std::condition_variable`, and so on. Pthread is a common threading library on POSIX systems. It features `pthread_create()`, `pthread_mutex_lock()`, and more. On Windows, you have `CreateThread()`, `WaitForSingleObject()`, and `EnterCriticalSection()`.

C# has `Thread.Start()`, `Task.Run()`, `Parallel.ForEach()`, and the `System.Threading` namespace.

Python lets you achieve multithreading using functions and modules like the `threading.Thread` class, the `multiprocessing` module, and `concurrent.futures` for concurrent execution of tasks.

Disabling all concurrency means forcing your program to run in a completely serial fashion. Depending on its architecture, this may be simple or almost impossible. With the use of a flexible task scheduler that allows you to run any task on the main thread, you're probably good to go. If your setup is based on spawning and joining dedicated threads in various places, getting that code to run on just the main thread will be tricky. You can use mutexes to serialize logic executed on different threads and thus emulate single-threaded behavior.

Watch out for any adverse side effects of disabling concurrency. For example, some functions may start parallel processing and wait for the

results. You'll have to work around this. Your program may also take a lot longer, and you'll have to live with that for this debugging session.

Random Numbers

Random number generators are often used to randomize some behavior on purpose. However, because computers are deterministic, these random numbers are technically faked, and the algorithms that generate them are called *pseudo-random number generators (PRNGs)*. They accept an initial value (called a *seed*) and generate a seemingly random new value from it. This new value is returned as the random one, but it also becomes the basis for the next random generation, so the algorithm produces a stream of values that each derive from the one before.

Choosing a good seed is essential to improving the randomness a PRNG generates. It's common to use the current date and time, thus ensuring the seed is always different: New random values will appear every time a user runs the program, as in the example below. However, if you fix the seed to be a particular hardcoded value, the PRNG will always deterministically generate the same sequence of results.

```
srand(time(0));  
printf("%d\n", rand());
```

The algorithm behind the PRNG greatly influences the quality of the numbers it generates. If you just need to produce a random display or selection, a simple algorithm will suffice. For security-relevant applications, however, you must turn to cryptographically secure PRNGs.

Similarly, finding areas of code that depend on a PRNG requires knowing what relevant functions are called. Standard methods of doing so exist in all programming languages, but you can always implement custom PRNGs. Try searching your code for `rand`, `random`, or `prng`. If you know the name of the one in your codebase, search for that as well.

TIP: LOOK OUT FOR PRNG-RELATED FUNCTIONS

In C, there is `rand()`. C++ has `std::random_device`, which is used to initialize specific engines (like `std::mt19937`) and distributions (like `std::uniform_int_distribution`).

C# has `Random.Next()` and `Random.NextDouble()` in the `Random` class.

In Python, you can generate random numbers using functions and modules such as `random.random()`, `random.randint()`, and the `random` module for various random number generation methods.

Two possible approaches can help you avoid the potential issues with PRNGs. One is to return the same values whenever your program requests a random number. The other is to use a fixed seed for each execution. This might mean just going with zero rather than the current time, but in other cases you may need to use a particular value to make your bug reproducible.

IN PRACTICE: BEWARE OF PRNGS BEHAVING DIFFERENTLY ON DIFFERENT PLATFORMS

I was once working on a program that procedurally generated landscape data, then used it to create features like villages, trees, and rivers. Issues with this generation arose when we brought the program to various platforms. We didn't catch them before the release because only certain initial values triggered them, so we ultimately had to rely on our users reporting problematic seeds so we could reproduce the problem and fix it.

If your program relies on PRNG behavior for essential functionality, you should be in control of the random number generation. Implement a published PRNG algorithm or reuse an open source one you can integrate into your code.

External Inputs

External inputs may also act as sources of randomness. User interactions via devices such as keyboards, mice, or touchscreens happen at unpredictable times with unpredictable results. (Some programs even use this approach to generate cryptographically secure random values.) Reading data from a file or external sensors for information can also produce randomness.

Since external input stems from a variety of sources, there's no one way to locate them. You can rely on standard functions to find input from users or files. If the source of randomness is custom functionality, however, it's more challenging to identify.

Try to avoid external inputs altogether when debugging. If you can reproduce your bug without manual user action, that's ideal. If not, try automating the process to avoid potential variations like when someone presses a button or for how long (see “Automate the Reproduction” on [page 29](#)). When working from external files, ensure they always have the same contents and metadata (time last accessed, modified, created, and so on). You can do this by implementing setup and teardown functions that automate file creation so you can be confident the environment is always the same.

Communication

A final potential source of unpredictable data is communication with the operating system or other programs. Irrespective of whether your program communicates with software on the same device (inter-process communication or system calls) or with programs on remote devices via a network, you can't foresee what data they will send you.

Locating functionality related to networking is relatively easy because standard functions can establish network connections, use them for communication, and shut them down. Any program communicating via a network has some layers of abstraction on top of the low-level primitives, but starting from there you can track down any dependencies and figure out what parts of your program depend on communication.

TIP: LOOK OUT FOR NETWORK-RELATED FUNCTIONS

In C on POSIX-compliant systems, you can find `socket()`, `send()`, `recv()`, `sendto()`, and `recvfrom()`. Windows provides similar functions with slightly different signatures. You may also want to look for usage of curl (<https://curl.se>) or Boost.ASIO.

C# enables network communication with functions and classes such as `TcpClient`, `TcpListener`, `UdpClient`, `WebClient`, `HttpClient`, and `NetworkStream` from the `System.Net` namespace.

When working with Python, network communication uses functions and modules such as `sockets`, `requests`, `urllib`, and higher-level libraries like `Twisted` and `aiohttp` for asynchronous network operations.

If it's at all possible, remove communication over the network, since you can never make it completely deterministic. However, when

networking itself produces bugs, you'll have to deal with this problem the way you deal with external input: Either always return the same message or play back a recording of previously recorded messages to create a deterministic sequence.

Put Determinism to Work

Assuming the randomness itself is not the bug you're looking for, but the bug occurs unpredictably, the root cause and the randomness must be related. If you identify the source, you can track down any data or logic dependencies from there.

You can also disable the randomness and force determinism to make your bug always reproducible. That lets you be more confident the bug is really gone when you can't make it happen, since you'll know it hasn't randomly vanished. Conversely, by forcing determinism that doesn't reproduce the bug, you can create a workaround that might unblock other people (see “Deploy a Quick Fix” on [page 201](#)). It might even serve as a permanent fix, as in the case of an uninitialized variable.

IN PRACTICE: OBSERVE RANDOM PATTERNS

When working on a 3D renderer, I encountered a bug that appeared only when the program simulated rain. However, the renderer randomly chose a weather state at launch and again after some time had passed.

It took some effort to track down the code that selected the weather on startup and the function that regularly chose a new state. However, when I had found those spots, disabled the weather change, and forced the initial weather to rain, I could debug the problem on every startup.

Disabling randomness grants you one more investigation opportunity: You can now perform both a “good” and a “bad” run and analyze how the randomness influences the rest of the program. Either debug your software (see “Step Through the Code” on [page 100](#)) or compare logs (see “Instrument the Program to Report Problems” on [page 62](#)) to find differences in behavior.

Consider Rare Hardware Issues

While computers are generally deterministic, there's still a tiny chance of actual randomness that can also randomly trigger bugs. Infamously, it's theoretically possible for cosmic radiation or high-energy particles to interact with computer memory and cause bit flips (corrupt the data stored in memory). RAM is particularly susceptible to this because it stores an electric charge, so high-reliability computing features error-correcting codes (ECCs) that detect and correct such bit flips.

Sometimes hardware components interfere with each other, components fail, or power supplies fluctuate. All of these can lead to hardware behaving nondeterministically or memory not being reliable. For example, gas-lift office chairs may cause screens to go blank for a second when the person sitting on the chair gets up, an effect caused by static electricity discharge. The chance for outside interference to the hardware is, however, negligible, so when looking into random behavior it's best to check any other possible sources first.

10. Ask Colleagues for Help

How? Ask your teammates or external colleagues for help. Always use an inquiry strategy that works for the person you're asking. Some people prefer more information upfront, others less.

Why? Other people are experts in their specific areas, and if your bug touches their area, they can help you significantly with minimal effort on their side.

With how big and complex software development projects are and how specialized everyone's role is, it's clear that you can't know everything yourself. When you ask the right person the right questions, they can give you relevant and helpful information. However, make sure to do your due diligence and spend ample time investigating your bug before asking. If your team doesn't have an expert on certain subjects, you can escalate outside of it. Often you don't even need a lot of information to make progress on the bug yourself. A tiny clue can make a huge difference. The relationship between a bug's symptoms and a spot in your program's sources is often unclear, but someone with the right knowledge of the program will be able to explain it so you can start digging deeper yourself.

In other cases, the bug may be more complex or distributed around the codebase, and you may need more help. If you've found someone with relevant knowledge, they might even give you the solution to the bug—or be able to take over the repair.

Ask Your Team

You can ask your team whether anyone knows anything about the bug using forums where the relevant people are already present. The daily stand-up meeting is the perfect opportunity for most teams, but a group chat can also work well.

Investigate First to Ask Better Questions

Investigation helps you learn about the program and its source code. Even if you don't find the cause of your bug, you'll have discovered information that can help you next time.

Also, the more you know about the bug, the better and more precise your questions will be, and the easier it will be to clarify the situation for your colleague and prevent them from retracing your steps.

Determine When to Ask for Help

There are a lot of factors involved in knowing when to stop doing your own investigations and ask for support. You can't know in advance how long it'll take you to understand the bug and where it comes from, and you can't know whether one of your colleagues has the exact information you need readily available. This is why you should give a short summary of your bug during convenient opportunities.

You certainly want to make sure to have covered the most obvious solutions to the problem. Make sure to try to find a shortcut first (see “Try Shortcuts” on [page 38](#)) and write down your progress (see “Keep a Debugging Log” on [page 140](#)). Before interrupting someone else, try to explain your bug and questions to an inanimate object (see “Talk It Through” on [page 173](#)). While it doesn't do any harm, and in a healthy team no one will frown upon your asking an obvious question every now and then, you should avoid doing so frequently. Your team will give your questions more attention if you save them for really difficult problems.

IN PRACTICE: LET PEOPLE ASK QUESTIONS

A recent graduate asked questions constantly when he joined our team. He became one of our best programmers very quickly, and I believe this is partly because he built up knowledge so quickly in the beginning. I've subsequently encouraged everyone to ask as many questions as they want. Of course, I might tell the people asking to go back and check various possibilities, but that's a helpful response itself because it gives them some hints about possible bug sources.

On the other hand, you don't want to wait too long before you ask for help, because you might spend a lot more time than necessary on the problem at hand. As a rule of thumb, spend at least 20 minutes investigating the bug yourself. If you're making some progress, keep going. If you have absolutely no idea and have been stuck the whole time, reach out to your team. It's better to risk asking too early and often than asking too late or not at all—you might get stuck for a long time fixing a bug you could have quickly solved with the right person's help.

IN PRACTICE: MAKE SURE THE RIGHT PERSON WORKS ON A BUG

I remember when a junior programmer on my team took way too long to ask for help. An expert in graphics programming was working on our 3D renderer to improve its performance after having written substantial parts of it, while the junior somehow ended up assigned to a bug about depth sorting being broken and went on a deep dive into the renderer to try and figure it out. The initial investigation took one or two days, while the expert had already fixed the problem locally after discovering it themselves. This bug could certainly have been fixed a lot more effectively if it had been assigned to the correct person at first. However, mistakes like this happen, and it's important to catch them by communicating openly.

Consider Your Target Audience

When you ask someone if they know anything about your bug, one option is to describe only the most basic facts, leaving out details or your previous investigation. You obviously can't go any deeper than this if you don't have any specifics, and the other person may know the answer from the basics alone. However, keeping your description simple also helps your contact stay open-minded, while sharing your line of thinking can bias them and block new ideas.

The other option is to provide as many details as possible. Tell the person you're talking to what you've tried, what you've observed, and even your current theories. Doing this risks distracting your colleague with too much information, particularly when you're speaking face-to-face and they can't double-check what you said later. You must also be very careful to distinguish your theories from facts.

The ideal balance between giving too little or too much information is, as so often with balancing opposites, impossible to find. You need to get to know your colleagues and their preferences, but they probably don't know offhand how much detail they need for a given case, so you must also learn to judge in the moment and adapt.

If the conversation gets stuck, you can trickle out information. Start with the facts you know for sure. Then share your guesses and suspicions until your colleague has something to contribute.

Be Approachable

Keep an ear open to other developers inquiring about your areas of expertise and give them pointers. Make sure you come across as approachable so that anyone on your team feels that they can ask you about your systems and learn from your experience.

If you have difficulty getting focused or back into the flow after interruptions, it's an excellent idea to communicate when you're open to interaction. People often use closed doors and headphones in the office to signal that they're busy, so make sure to spend some time without them, and indicate openness in your chat program's status feature if you're working remotely.

Because it disrupts your flow and you may have to investigate a bit yourself, you may feel like giving people advice takes time away from your actual job, but helping colleagues *is* part of your responsibilities and the workflow of functioning teams includes it. Your expertise means you'll probably be able to point your co-worker in the right direction quickly, whereas investigating the problem themselves would take much more time. While you're at it, you can even give them some hints about potential pitfalls in the system and thus kickstart further investigation. In the long run, your colleague will learn more about the program's systems and not need as much assistance.

There's another angle to this if you've written a system. When other developers repeatedly ask for help about particular bugs or modifications, they're giving you first-hand information about what you can make clearer in your next design or implementation. If you don't have the time to update the program, consider documenting the shortcomings or creating a FAQ you can share with anyone asking for help.

Ask for External Support

While external experts might not be as easily accessible, they may still be able to help with your problem. You might find them in online forums, social media communities, or support portals. Getting to the right person will take more time when going outside your organization but may very well give you access to even deeper knowledge.

You should start by asking through the most specific channel available to you, then broadening your range. While the code's original author is the best source of help, community support for popular technologies is often widespread, and enthusiasts may know more than the original author about specific use cases.

If you're using an off-the-shelf framework such as Electron, Qt, or Unreal Engine, you should start by asking on that framework's support channel. If you're using an open source library from GitHub, try posting on the library's issue tracker. If you're using proprietary third-party libraries, ask their support staff.

When a specific technology has no related communities, go more generic. For example, try a community for the overall type of technology you have trouble with. Various Stack Exchange communities and subreddits cover specific topics and allow you to ask questions.

For strictly programming-related issues, you can also go to the even more general Stack Overflow. You may not get any answers for your particular use case but may still find some advice that's relevant to your problem.

Write a Great Bug Report

As covered in "What Makes a Great Bug Report?" on [page 11](#), describe all the symptoms you're seeing and compile all the data you can get on the issue's behavior and environment. In this case, it's better to err on the side

of giving too much information than too little because it's more difficult for someone external to ask you for further details.

Keep in mind that the people you're asking don't have the context that you, your teammates, or even your more general co-workers would. They know literally nothing about your program and the technology you're using. You need to provide all this information.

WARNING

Remember that you must abide by any nondisclosure agreements and ensure you don't accidentally disclose any non-public information about your program. Especially if you're on public forums, consider posting under a pseudonym that can't be linked to you or your company.

When dealing with the original developer, the best way to get help is to supply a minimal reproducible example (see “Build a Minimal Repro” on [page 210](#)). Developers are often strapped for time and must prioritize severe and frequent problems. If you're the only one exposed to a particular bug and it takes significant time to investigate and fix, the developer may never get around to it.

Expect replies to take a while, depending on the bug's importance, the fix's complexity, and the creator's availability. The original developer may not be very transparent about their progress, to the point where they release a new version of the software that magically includes a fix for your bug without ever having told you about it.

Your initial support request may be assigned to someone who will triage it to figure out if it's an actual problem with their software or if you're making a mistake. They'll probably also try to rule out the good old *turn it off and on again* strategy. Of course, it's a good idea on the third-party company's side to shield their developers from trivial support requests, but for real bugs, this shielding slows down the process of getting a proper fix.

If your first contact can't easily reproduce your issue, you must be insistent, especially if you're convinced your problem comes from an actual bug on their side. Keep pushing until you get to the actual developers who can help you. Fixes may also come in surprising or inconvenient ways, from incorporation into scheduled updates to custom patches that you must

apply to the software's source code and then rebuild it yourself. If you're lucky, the developer will still incorporate that change into their own codebase later. If you're unlucky, this fix is specially tailored to you, and you have to keep applying it to future software versions. For cases where you don't have source code, you may receive a unique build and again have to hope that the fix will also appear in future regular releases.

Raise Attention

Being employed by a renowned company and making that visible when asking for help will probably speed up a reply to your support request. Developers largely care more about popular applications or games that use their technology, where many people will encounter bugs if they exist.

Even if you don't work for Microsoft or Larian, it can be a great idea to ask about your problem on social media. (You should do this only if you can talk about your project openly or at least share that you are working on something relevant.) This way, you're both contacting the community as a whole and potentially getting direct support from the original developer. You can increase the probability of getting a reply by tagging the company or developers it employs, or by using a relevant hashtag.

Openly asking can put more pressure on the original developer compared to asking in private. Hopefully, they'll want to provide excellent support and put in some extra effort to help you quickly.

Leverage Your Network

Getting through to the right people to investigate and fix your bug can be difficult. Keep in mind that some software companies have tens of thousands of developers. Ideally, you have a champion who is internal to the company you're dealing with and relentlessly ensures that your problem gets to the correct person. You can find a champion by pure luck when contacting the company, but you can increase your odds by leveraging your professional network. It'll undoubtedly be a lot easier to convince a pre-existing personal connection that you need help, that your problem is real, and that an internal developer needs to address it. You must be sure that the last factor applies, though, so you don't strain your relationship.

This advice isn't helpful if you don't have a network in the industry yet, of course, but asking for help can build on even a tentative relationship.

Let's say someone from the developer company gave a talk you attended. The presenter probably put their email address on their slides, giving you a direct form of contact, and the talk itself gives you common ground. Email them, tell them what you liked about the talk, and segue into your problem. This approach can be the first step to both getting help and building your network. The person you're asking might even be grateful for your information because it's input from real-world users, which improves their product. Even if they aren't, they've now read your name and will hopefully recognize it in other contexts.

An alternative to using email is to reach out to contacts on professional networking sites such as LinkedIn. Looking for contacts of your direct contacts may provide you with a list of people who could help you. Either reach out yourself, mentioning your shared contact, or ask your contact for an introduction.

IN PRACTICE: REACH OUT TO PEOPLE YOU DON'T KNOW PERSONALLY

Our OpenGL renderer once messed up meshes—even simple cubes—only on Nvidia graphics cards. After discussing the bug with several people on the team and not making any progress at all, I remembered that one colleague had recently gotten an Nvidia engineer's business card after a conversation!

In a desperate Friday night email, I sent this engineer an explanation of our problem, including the minimal reproducible example. I went home and didn't think too much about it anymore. When I returned on Monday morning, I had a reply waiting in my inbox with a thorough explanation of what we had been doing wrong. (Of course, it wasn't Nvidia's fault. We had been providing the wrong vertex stride for our meshes, and all the graphics cards from other manufacturers had been simply ignoring our mistake, but Nvidia was following the OpenGL standard.)

Consider Reply Durations

Getting a reply from someone outside your company can take anywhere from days to weeks or even longer, so it's best to reserve for cases when the problem isn't urgent or when you have no other options left. The delay will also give your mind some rest, since you've at least temporarily handed the situation over to someone else, which might in turn spark fresh ideas and new perspectives.

11. Search in All Files

How? Use tools to search through your source code, assets, and binary data. You can look for strings you've encountered in your program's user interface or elsewhere in the data or code. Try partial strings as well because they're more likely to give results if code concatenates text.

Why? Developers often use text to connect various program pieces, and software often produces it as well, which lets you trace its origin in the code. Even binary formats contain text fragments that can give you valuable information.

Choose and Configure Your Search Tool

You have a few different options for searching files. All can be useful, depending on your situation, but they have a variety of trade-offs.

Your integrated development environment (IDE) is the tool you most likely use to develop your program. It knows about all the source files and their organization and has analysis features that go beyond text, such as searching for specific symbols, filtering by type of symbol, and narrowing the scope to specific projects. Because you use an IDE to develop your program, it's also readily available.

TIP: USE VISUAL STUDIO'S CODE SEARCH

In Visual Studio, press CTRL-T to open the tab for a powerful code search. Prefix your term with `f` (files), `t` (types), or `m` (members), respectively, to limit your search to only one of these kinds of entities.

The line between text editors and IDEs has become blurry, with some popular editors adopting more and more IDE-like features. In general, though, a regular text editor is more focused than an IDE on editing individual files but often includes powerful search-in-files features.

Specialized tools for searching file sets are even more detached from your development environment than text editors, but therefore even better suited to files that don't relate to the program's source code. The most

commonly used such tool is `grep`, which has been around for a long time on Unix-like systems and has a powerful feature set. Relatedly, `ripgrep` (<https://github.com/BurntSushi/ripgrep>) is an alternative more focused on performance. Windows also provides tools like FileLocator Pro (<https://www.mythicsoft.com/filelocatorpro/>).

Even more specialized tools search code specifically. These include classic Unix tool `ack` and the more recent `ag` (<https://github.com/ggreer/the-silver-searcher>). VsChromium (<https://chromium.github.io/vs-chromium/>) is an extension for Visual Studio that offers a set of powerful tools to search source code. Its main differentiating feature is keeping all source files in memory, allowing extremely fast full-text search.

If you're searching for a file with a specific name, tools such as Everything (<https://www.voidtools.com>) or `fzf` (<https://github.com/junegunn/fzf>) generally outperform operating systems' built-in options. They provide fuzzy search over complete filepaths, so you can easily find specific files even if you only vaguely know their names.

Choose the Right Configuration Options

Whatever tool you're using, pay attention to any configuration options. Some tools may default to not searching a particular set of files or ignore large or binary files altogether. As “Peek into Nontext Files” on [page 89](#) discusses, these files may be the ones you need. On the other hand, reducing the number of files searched is the best way to make the process shorter. If you're looking for matches in C++ code, include only `.c`, `.cpp`, `.h`, and `.hpp` files, for example. Another way to filter includes only code files that are part of a given project or directory.

How quickly a tool can read your files makes a massive difference in performance. Having the files in memory is the best option but isn't always feasible. An SSD, which provides excellent seek and read speeds, can offer balance.

Choose Useful Search Terms

Most tools offer different modes or options when searching. You can decide whether you're performing a conventional search for a string or using regular expressions. When using normal text, you may also be able to specify whether the query is case-sensitive or includes only complete

words. It's generally best to start with a broad approach where any substring is a valid result and the case doesn't matter. If this gives you too many results, you can start enabling more restrictive options.

While regular expressions can be tremendously valuable, they're also quite complicated. If you know how to compose them, you'll easily be able to find relevant results that a normal search wouldn't uncover and set up more flexible patterns if you're not quite sure what you're looking for.

TIP: LEARN REGULAR EXPRESSIONS

Regular expressions can become very complex, but you can start with simple patterns:

`ColorGr[ae]y` Matches both `ColorGray` and `ColorGrey`

`0\+1f` Matches `0.01f`, `0.001f`, and any arbitrary number of `0s`

`0\.0*1f` Alternatively matches `0.1f`, `0.01f`, and any arbitrary number of `0s`

`strn?cpy` Matches both `strcpy` and `strncpy`

`== '.'` Matches any comparison with a single character

`printf\(.*\)` Matches any call to `printf()` independent of the supplied parameters

`Set\w+Position` Matches any function referring to a position described by word characters (letters, digits, or underscores), like `SetAbsolutePosition` and `SetRelativePosition`

Note that `\` is used to escape characters with special meaning.

Tools like Regex101 (<https://regex101.com>) and RegExr (<https://regexpr.com>) make it easier to build and learn about regular expressions.

Use Terms from the Program's Output

If the software prints any interesting information about what it's doing or even a warning or error related to the bug you're working on, those strings may very well lead you to the right spot. The log output might contain the name of the system or even the function name it comes from. If it doesn't contain any identifying information about its source, you can still take fragments from the message and search your code for those (see "Look for Strings Composed in Code" on [page 90](#)).

Text in other areas, like GUI elements, network messages, or serialized files, may also be useful. Since your program must have generated this text,

at least some of it may lead back to the software's sources and therefore areas relevant to your bug.

Try Source Code Names

You could also use the functionality you're debugging to guess what the code might call a given feature. For example, if your bug has to do with input handling, the file or function name might include *input*, *controller*, *touch*, *keyboard*, *mouse*, or *gamepad*. While you have a number of different potential terms in each case, you should be able to quickly rule out any that aren't related after all.

Strings you encounter during earlier debugging steps can also make good search terms. Source code may reference other data via filenames and IDs. You can look up this data (that is, open the file in question) and find references back to the source code.

Remember, however, that any text the user sees may not appear verbatim in the code. For instance, it may have been translated.

Search Beyond the Source Code

Now that you know how and what to search, the next question is where you should do it. "Everything, everywhere, all at once," might be ideal in one sense, but could take a while, provide too many results, or even be impossible without specialized tools or access privileges. Therefore, you must narrow your search area.

As discussed previously, searching the program's log output is straightforward, and it may be available in your terminal's buffer or a logfile. Source code files are also easy because they, too, are text files and present in your IDE. Your program's assets may be more difficult if they're binary but may reveal worthwhile information.

Software configuration and settings are often in text format as well. Be careful when trying to find them because many programs store their configuration in a central location, whether a dot directory in your home directory (Unix-like systems), in your user's *AppData* directory (Windows), or even in the registry (Windows as well). Such directories are hidden by default, so you need to do some work to track down the correct ones.

Databases vital to your program's execution may also hold valuable information. Their contents are often encrypted, compressed, or stored in an

optimized fashion, though, requiring specialized tools for search.

On a higher level, you can search through your project’s metadata. Try to find relevant information in your project management tool. It may contain data about tasks related to relevant feature creation or references to previous, similar bugs and their history.

Your project’s documentation could provide further information about the system. It may offer background information about how it came to be, its limitations, and maybe even a list of known issues. Similarly, your version control system or your own notes may contain relevant information from the project’s history (see “Check Version Histories” on [page 52](#) and “Keep a Debugging Log” on [page 140](#)).

Ultimately, you can search the internet, which can be useful only if you’re dealing with something others use as well (see “Search the Internet” on [page 47](#)).

Peek into Nontext Files

The line between text and binary files isn’t as clear as you might think. Just as editors can garble text files in certain types of encoding, binary files aren’t necessarily completely unreadable. Existing or custom tools can help you find information in files with formats like *.zip*, *.odt*, and *.png*.

Even without knowing a file’s format or parsing its binary data, it’s possible to find text fragments, like raw strings encoded in ASCII or UTF-8. (Some text editors may need a little bit of nudging to make them open a file or set the right encoding.) If you can see these fragments, the previously described find-in-files tools can locate them.

TIP: FIND FILENAMES IN .ZIP FILES

If you open a *.zip* file in a text editor and scroll all the way to the end, you’ll see a lot of weird characters and what appears to be garbled text. Between these unsupported characters, though, you can also spot the full paths and names of the files that the archive contains.

Hex editors display a file just like it is, byte-by-byte, making them indispensable when you’re dealing with binary files. By viewing the bytes as text, you may be able to figure out how a file relates to your bug.

Look for Strings Composed in Code

If you're looking for a string in your source code, keep in mind that text is often composed of partial strings. That means the text you see from your program may not be present in that exact form in the code. Here are a few examples in different programming languages:

- C: `printf("Hello %d\n", rand() % 100 + 1);`
- C++: `std::cout << "Hello " << (rand() % 100 + 1) << std::endl;`
- C#: `Console.WriteLine("Hello " + new Random().Next(1, 101));`
- Python: `print("Hello {0}".format(random.randint(1, 101)))`

If your program was to output `Hello 42` and you searched for that literal string, you wouldn't be able to find anything. Focus on partial strings instead. Make sure to leave out anything that looks like it could have been composed in code. In the example, searching for `Hello` would take you to the correct spot, but maybe also way too many others. This could also be an opportunity to put your regular expression skills to the test (`Hello \d+`).

It takes some experience to reliably identify which parts of a string have been composed. (Concrete specific file or function names, text that the user previously provided, and anything that changes over time, such as the date and time, get composed into complete messages.) Sometimes, it's easier to search for a short substring than guess. Use an uncommon string so you don't get any false positives. In the case of printing `Hello` and a random number, searching for `("H` would have uncovered most spots in the source code.

Use Searchable Patterns in Code

Think carefully about names during development. The names you give your variables, functions, and types are the same names you or someone else may end up searching for later. This is also true of the data you put into your configuration files or assets.

Standard programming practices promoting code quality call for descriptive and meaningful names. Those criteria generally make a name easy to search, but you should still try to be as unique as possible when

picking one. For example, `Pos` as a variable name is tough to search for in any meaningful way because it is also a substring of many other words. `Position` is a lot better. If it refers to an image, `ImagePosition` is even better and most likely doesn't collide with any unrelated areas in the code.

You can't always make a name unique in the code, so you may have to resort to putting searchable text in comments. There you're free to do whatever you want as long as your team doesn't complain. You can come up with text that lets you easily return to specific spots, for example, `// TODO-yourname`.

Some projects require you to highlight changes in existing code. In such cases, you should choose even more unique markers that attribute the change to your team and the individual author and describe the reason you made it. Consider something like this:

```
// @START - teamname - yourname - Reason for change
--snip--
// @END - teamname - yourname - Reason for change
```

This kind of step could be necessary for projects that are shared or transferred without intact version control history.

12. Collect Crash Dumps

How? Implement support for crash dumps yourself or leverage existing third-party libraries. Once you've done that, you should also set up a system that lets you access files your users generate.

Why? Crash dumps let you debug crashes that occur at any time and place, such as those your users encounter on their devices. Overall, there are several ways to log information relevant to bugs and crashes, such as telemetry data. In fact, server-side services often record how many requests fail and investigate when the error rate is abnormally high. However, this section focuses on crash dumps in particular.

When your program unexpectedly fails, it should ideally trigger the creation of a crash dump. You can examine that file later on any computer

without access to the device where the crash occurred. Any build you give out to testers or users should come with a crash dump feature.

Because crash dumps allow catching bugs in the wild, they're handy for gathering information about those that occur only rarely or that potentially happen only outside of the development environment. The crash dump contains information about the state of your program when it failed, like the call stack, variables, and memory.

Trigger a Crash Dump

Most crashes come from hardware exceptions, which means the hardware wanted to prevent the program from executing. These are problems that you can't and should not recover from. The most important types are division by zero or illegal instructions (pretty much what the names say) and access violations, where your program tried to read or write a memory address to which it lacked access.

Both access violations and illegal instructions typically occur when a pointer indicates the wrong address. Accessing a null pointer will also trigger an access violation, while trying to call a function with the wrong address will likely cause an illegal instruction.

Programs may also trigger exceptions. On Windows, the `RaiseException()` function will ultimately prompt a crash dump if the exception isn't handled elsewhere. Such a mechanism is often used to let assertions trigger crash dumps (see “Instrument the Program to Report Problems” on [page 62](#)).

Get Information from a Crash Dump

Acquiring a crash dump for a bug you're working on is almost as good as catching it in the debugger yourself. You can load the file into your debugger and “run” it there to get most of the same information (most importantly the call stack). Some crash dumps may contain variable values and program memory, which is hugely valuable but increases file size. If yours doesn't, you have to go by the logic's state and guess which variable may have had which value.

The biggest drawback of crash dumps is that they're not interactive. You can't see how the logic evolves over time. If you were actually debugging, you would potentially have stepped through what came before,

and you might be able to continue or manipulate the instruction pointer to hop over the problematic code. However, if you had only caught the hardware exception when it had already been thrown, you wouldn't be able to continue past it anyway.

What you see in the debugger may also be challenging to analyze because the displayed code isn't necessarily what's being executed. Crash dumps often stem from end-user builds created with optimizations enabled, which means that the compiler transformed the code quite a bit. If you repeatedly observe crashes in a specific function but can't figure out what's going on, it may be worthwhile to disable optimizations for that specific function and ship a build with this change (see “Step Through the Code” on [page 100](#)), being careful about resulting performance issues.

Implement Crash Dumps

First, you must decide the right time to introduce crash dump support to your program. Since the goal of doing so is typically to catch users' issues, it may happen only shortly before release.

However, having crash dumps in place early during development is also hugely beneficial. It makes it easier for you to weed out crashes before the release and allows your testers to provide more hard data when creating bug reports (see “What Makes a Great Bug Report?” on [page 11](#)). Having too many crashes during development also slows down the whole team.

Adding support isn't much work in the sense of writing a lot of code because it doesn't require you to create complicated new systems, define data structures, or implement complicated algorithms. The complexity comes from dealing with highly platform-specific features that vary between operating systems and hardware architectures.

Additionally, you're operating outside the realm of functionality a program typically tackles. You have to write code that can run after your program has caused a catastrophic error and work flawlessly when there's not much in the environment it can rely on.

Implement Crash Dumps for Your Platform

Implementing crash dumps is highly platform-dependent, with significant differences between various operating systems on PCs, mobile devices, and proprietary devices. This section shows an example using C on Windows.

The first step is to react to the crash. Very early on in your main function, you have to set up a handler for unhandled exceptions:

```
int main(int argc, char** argv) {
    SetUnhandledExceptionFilter(handle_exception);
    --snip--
```

The `handle_exception()` function implements the creation of the crash dump, which also conveniently stores the data in a file:

```
#include <windows.h>
#include <Dbghelp.h>
#include <tchar.h>

typedef BOOL (WINAPI *MINIDUMPWRITEDUMP) (HANDLE hProcess,
    DWORD dwPid, HANDLE hFile, MINIDUMP_TYPE DumpType,
    CONST PMINIDUMP_EXCEPTION_INFORMATION ExceptionParam,
    CONST PMINIDUMP_USER_STREAM_INFORMATION UserStreamParam,
    CONST PMINIDUMP_CALLBACK_INFORMATION CallbackParam);

LONG WINAPI handle_exception(struct _EXCEPTION_POINTERS* exception_info) {
    HMODULE lib = LoadLibrary(_T("dbghelp.dll"));
    MINIDUMPWRITEDUMP pDump =
        (MINIDUMPWRITEDUMP)GetProcAddress(lib, "MiniDumpWriteDump");

    HANDLE file = CreateFile(_T("mydump.dmp"), GENERIC_WRITE,
        FILE_SHARE_WRITE, NULL, CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL, NULL);

    _MINIDUMP_EXCEPTION_INFORMATION ExInfo;
    ExInfo.ThreadId = GetCurrentThreadId();
    ExInfo.ExceptionPointers = exception_info;
    ExInfo.ClientPointers = FALSE;

    pDump(GetCurrentProcess(), GetCurrentProcessId(),
        file, MiniDumpNormal, &ExInfo, NULL, NULL);
    CloseHandle(file);
```

```
    return EXCEPTION_CONTINUE_SEARCH;
}
```

Now that you have the crash dump in a file, you'll be able to load it up in a debugger and look at what has happened. As it is, the process involves asking the users or testers to share the file with you for every incident.

An ideal solution would share it automatically. However, the more logic that must run in the event of a crash, the more can go wrong. Imagine your program operates on a device that runs out of memory, and the memory allocation function returns a null pointer. Because you don't check the result but just write to the provided address, your program crashes. In this common scenario, you'll very quickly encounter an access violation.

Having to run extensive logic in a catastrophic event isn't a great idea. That's why there's a more reliable but also more complex solution: You can implement your crash dump handler as an out-of-process handler. Instead of doing any extensive computing, the crashed program delegates the crash handling to another process. This may spawn when a crash takes place or run continuously as a kind of watchdog. The latter option simplifies response when a program does fail, because you can fully prepare the handler and have all the memory it requires reserved in advance. Then it just needs to create the crash dump and perform any further processing. (It would also be nice to implement a little UI where the user could describe what they were doing when the crash occurred.)

Generate and Store Symbol Files

The significant advantage of crash dumps is that they work with end-user builds, but those are typically *stripped builds*, meaning the executable no longer contains any debugging information. You won't see any variable names, function names, source filenames, or line numbers, but only assembly instructions and memory addresses. That's still better than nothing, but it can be very hard or impossible to figure out what part of your source code is to blame when you have such limited information.

While your executable should get stripped during the process of preparing a distribution build, you should also make sure to generate a symbol file that solely contains the otherwise missing debugging

information. You may even have multiple symbol files for a single build, typically one for each executable and one for each shared library (*.dll* or *.so* files). You must archive these so you have them readily available if you need to debug a crash dump.

One symbol file is only ever associated with one particular build, so make sure you know which belongs to which. Assuming that the builds you distribute have a version number, a working simple approach is to label a directory accordingly and store the files there. Symbol files contain whatever information is needed to correlate the code and data in the executable with the source code. For example, the *.pdb* format used by Windows contains the following:

Symbolic information Function or variable names and their memory locations

Source code information Filenames and paths, line number information (mapping memory addresses to specific lines of code)

Type information Structure layouts, function signatures, type definitions (enums, typedefs, and user-defined types)

Build information Compiler settings, version information, and timestamps related to the build process

When debugging a crash dump, you may see that parts of the call stack are inside a third-party library. When you don't build libraries yourself, you need symbol files to debug them effectively, so it's standard procedure to ship them together. Even Microsoft does it. Symbol files can get quite large. Because they contain a lot of text, they compress well, but you have to take care of the compression yourself. Even the compressed version can still take up several gigabytes, which you'll have to store for each release until you're sure you won't have to deal with its crash dumps anymore.

IN PRACTICE: WATCH OUT FOR EXCESSIVE FILE SIZES

One project I worked on used a third-party framework that consisted of multiple *.dll* files. It came with multiple plug-ins, each built as one *.dll*. The symbol files for all of them exceeded 60GB in total, and storing this for each nightly build was just way too much. We decided to store only the symbols for our executable and exclude those of any third-party libraries.

Choose the Right Type of Crash Dump

Most crash dumps contain exception information (the reason for the crash), a call stack that includes registers, a list of threads, a list of modules, metadata about the CPU and operating system, and any active operating system handles (references to files, synchronization primitives, processes, and so on). They also include memory, which contributes the most to their size. You can leave out all or some of it, but you may inadvertently exclude important information by doing so.

On Windows, you can choose between a *full dump*, which includes all the accessible memory, or a *mini dump*, which only contains memory that local variables on the stack directly or indirectly reference. You can specify what data you want when calling `MiniDumpWriteDump()`.

So far, all of the discussion has been about CPU-side issues, but graphics processing unit (GPU) crash dumps also exist on some proprietary systems. Nvidia's GPUs, for instance, offer Nsight Aftermath (<https://developer.nvidia.com/nsight-aftermath>).

Turn Crash Dumps into Actionable Data

Once you implement crash dump creation, you should look into access. Make it as easy as possible to get the most information from your users. At the very least, the program should tell them that a crash occurred and the location of the crash dump. It could do this the next time a user starts it after a failure.

Your software should also share those crash dumps automatically with you. Shared network directory storage is straightforward and lets you regularly pick up the files.

A better solution is to upload them to a shared online service. This solution works for anyone independent of where they are in the world and which network they're on.

Once you get an online service that collects your crash dumps, you have many more possible options than just storing them. The service can analyze your files and extract information so it can display them conveniently and you can avoid opening each one just to see where the program failed. Note that the online service also needs access to associated symbols for each executable.

The next step is to group crashes that happened in the same or a similar place. That gives you more data for each bug and lets you draw statistical patterns, such as which crashes occur most often, how stable each release is, and whether a problem recurs after being marked fixed.

These kinds of statistics will improve your program's overall quality by helping you focus on important issues. Looking at the call stack and crash frequency in a dashboard makes your debugging process easier.

To facilitate this workflow even more, you may want to add additional information such as the program's log (see "Instrument the Program to Report Problems" on [page 62](#)) or let the user specify what they were doing when the crash occurred. You should track information like the CPU type, graphics card, and amount of installed RAM. Status reports like the amount of free memory at the time of the crash can also be helpful. Patterns from all this data could reveal information about the bug's cause, indicating factors such as out-of-memory problems or specific CPU issues.

Use Third-Party Tools to Simplify Crash Reporting

While the basic implementation of crash dumps is relatively straightforward, it can quickly become complex if you want one that works across all your supported platforms, collates reports, and provides actionable insights. Since you don't often use crash reporting as a selling point, it's rare to spend much time on it.

Luckily, third-party libraries and services can take most of the complexity off your hands. Such solutions can have a wide variety of feature sets. Some only generate crash dumps, while others offer full-fledged online features that provide lots of statistics and insights.

The very first step you should take is checking whether your target platforms already offer such services. Quite a few proprietary platforms do. They'll have the most knowledge about their low-level components and access to data and features that might not be available to you.

Windows Error Reporting (WER) can generate a crash dump for any program (<https://learn.microsoft.com/en-us/windows/win32/wer/windows-error-reporting>) and automatically upload it if you've signed your software properly. Getting access to these reports can be a bit tricky, though.

Open source libraries can also take most of the work off your hands. Google has two: the bare-bones, in-process Breakpad (<https://chromium>

[.googlesource.com/breakpad/breakpad](https://github.com/google/breakpad)) and Crashpad (<https://chromium.googlesource.com/crashpad/crashpad>), which generates crash dumps outside processes and facilitates uploading and sharing.

More full-featured commercial services are available as well, such as Backtrace (<https://backtrace.io>), Firebase Crashlytics (<https://firebase.google.com/products/crashlytics>), and Sentry (<https://sentry.io/>).

Summary

By now you've tried a range of ways to find where your bug lives: educated guesses based on experience, internet and codebase searches, a trip through your version control history, the debugger, and a conversation or two with people who know the code. Any one of them might have handed you an entry point, the specific spot in the source worth digging into. With that spot in mind, you're ready to stop searching and start examining. [Part III](#) is where you look closely.

Part III

EXAMINE



In the last part, you probed your program and found an entry point, a specific spot worth a closer look. Now the work changes. Probing was about quickly narrowing down where to look; examining is the slower job of studying that one spot until you understand exactly what's going wrong and why.

Examining and probing are closely linked, and in practice you'll move back and forth between them. A close look at the code may raise a new question, so you'll probe for where to look next, then settle in and examine again. You don't have to march through the workflow in order. You'll loop between these two steps for a while, drop back to reproduction ([Part I](#)) when you need to, and jump ahead to a fix ([Part IV](#)) when you think you've found one.

The strategies in this part start with your program's source code, as that's where bugs most often hide and where the best examination tools are aimed. From there, they move to changing the program to learn more, bringing in outside help when you're stuck, and keeping notes so you don't lose your way. If none of this takes you to the root cause, you can feed what

you’ve learned back into the probing strategies from [Part II](#) and go around again.

13. Step Through the Code

How? Run your program inside the debugger and use its tools to analyze what’s going on. Step through the code, use breakpoints, and carefully consider the information the debugger provides.

Why? The debugger allows you to observe your code’s execution and examine potentially relevant spots in detail. You’ll see how everything unfolds in a concrete use case. Pay close attention the whole time to anticipate problems in the code you’ll step through next and remember roughly what happened if something goes wrong. It also helps to keep an eye out for what effect other control-flow decisions could have and whether any might lead to your bug (see “Read the Code” on [page 113](#)).

NOTE

Even if a bug’s root cause doesn’t lie in the code, the debugger can provide valuable details about how a mistake elsewhere affects the program.

You should always have the debugger ready (see “Attach the Debugger by Default” on [page 59](#)), but don’t get bogged down using only the debugger either. Keep an open mind and try other strategies as well. If you see suspicious behavior using the debugger, switch to other strategies to gain more insights, then debug your program with that knowledge as your input.

Choose the Right Build Configuration

You can generally compile code with different settings to influence the build process and yield varying results. These settings usually affect the program’s performance and debuggability because they direct which optimizations the compiler applies and which compile-time options it configures. Your IDE or build system generally groups such settings into configurations to help you specify what you need.

The two most common build configurations are debug and release, and, luckily, these names are pretty descriptive. You should be using the debug build configuration when you're debugging. The release build configuration produces optimized builds that are much harder to debug. The following sections describe why that's the case.

Mind the Effects of Optimizations

In most editors, the line is the atomic unit of code to which you can assign a breakpoint. It's also the smallest step you can take through the code using the debugger. However, this is a little removed from what the CPU executes, which is why debugger and breakpoint behavior can surprise you.

Building consists of many steps, such as compiling and linking, that translate your code into individual assembly instructions and optimize it. Both actions together lead to the disconnect between your line of code and the instructions the debugger operates on. One line generally translates to multiple assembly instructions.

For example, look at this line:

```
for (int i = 0; i < 10; ++i) {}
```

Using Clang without optimizations, this produces the following assembly code:

```
        mov     dword ptr [rbp - 4], 0
.LBB0_1:
        cmp     dword ptr [rbp - 4], 10
        jge     .LBB0_4
        jmp     .LBB0_3
.LBB0_3:
        mov     eax, dword ptr [rbp - 4]
        add     eax, 1
        mov     dword ptr [rbp - 4], eax
        jmp     .LBB0_1
```

Or check out an innocent-looking line like this:

```
SomeClass some_instance;
```

The result is a lot more complex due to the definition of `SomeClass`:

```
class SomeClass {
public:
    int x = 0;
    int y;
    SomeClass()
        : y(1) {
    }
};
```

It translates to:

```
    lea    rdi, [rbp - 8]
    call   SomeClass::SomeClass() [base object constructor]
```

```
SomeClass::SomeClass() [base object constructor]:
    push   rbp
    mov     rbp, rsp
    mov     qword ptr [rbp - 8], rdi
    mov     rax, qword ptr [rbp - 8]
    mov     dword ptr [rax], 0
    mov     dword ptr [rax + 4], 1
    pop     rbp
    ret
```

As you can see, if you step through the original C++ source code on a per-line basis, you will skip quite a few instructions at once. The problem is that each one may trigger the bug you're looking for.

IDEs usually have the option to show you the generated assembly code during debugging. If you enable this view, you can step through the actual instructions as the building process created them. Such a view gives you much more fine-grained control and may even be necessary to pinpoint an issue.

Code optimization also messes with your lines of code in many different ways, depending on which options it uses. With optimization

disabled, the compiler will translate your code to assembly on a one-to-one basis. At the lower levels (`-O1` or `-O2` compiler options), it will optimize your program safely. The maximum optimization level (`-O3`), however, may break your code.

Here are some common optimizations that make debugging harder:

- Eliminating common subexpressions to avoid repeated calculations
- Combining multiple loops to reduce overhead
- Moving code out of a loop if the program doesn't depend on it to avoid redundant computations
- Reducing the overhead of function calls by inlining functions
- Rearranging control flow to improve branch prediction
- Generating vectorized code to make use of SIMD instructions
- Reordering memory access to minimize cache misses

For example, look at this function definition:

```
int f() {  
    int x = 0;  
    int y = 1;  
    return x;  
}
```

Using Clang without optimizations, this translates to:

```
f() :  
    push    rbp  
    mov     rbp, rsp  
    mov     dword ptr [rbp - 4], 0  
    mov     dword ptr [rbp - 8], 1  
    mov     eax, dword ptr [rbp - 4]  
    pop     rbp  
    ret
```

If you turn on optimizations (using `-O2`), it becomes:

```
f() :  
    xor     eax, eax  
    ret
```

Enabling optimizations might move whole sections of code around. If you try to debug such optimized code, your IDE's execution pointer can jump around wildly when trying to step to the following line, skipping several lines or even going backward. Execution might never reach several lines because the optimizer recognizes them as superfluous and removes them. The optimizer can also summarize multiple lines into one instruction.

The same goes for breakpoints. If the optimizer removes a line with one, some IDEs try to be helpful and quietly move the breakpoint to the next line that has generated instructions, but that's, of course, not necessarily what you want. If you're trying to debug this one line and your breakpoint gets moved past a `return` statement or into an `if` condition, it might never be hit.

Watch Out for Build-Specific Behavior

Compile-time options may also greatly influence how the debug or release builds behave. These options let code react to the build configuration and change its logic accordingly, potentially swapping out complete implementations or maybe just subtly tweaking some behavior. For example, it's common to have more safety checks and log output in debug builds than release builds.

Both optimization and arbitrary behavior change between different configurations, making your debugging job more difficult. Differing behavior can result in bugs not occurring in debug builds at all or, alternatively, only in debug builds. For example, the debug runtime may be filling all newly allocated memory with a specific pattern to make it clear that this memory block hasn't been written yet (see "Instrument the Program to Report Problems" on [page 62](#)). While that's useful to detect vacant memory, it also makes bugs potentially invisible until they show up in release builds because those memory blocks have random contents.

Similarly, bugs commonly don't appear in debug builds that literally interpret code but show up in optimized builds where the compiler applies all the assumptions the language specification permits.

TIP: BEWARE OF C++'S UNDEFINED BEHAVIOR

C++ compilers infamously incorporate the language's undefined behavior when optimizations are enabled. Take this code as an example:

```
#include <stdio.h>
int main() {
    int i = 1;
    printf("Hello\n");
    while (i > 0) {
        i++;
    }
    printf("Goodbye\n");
    return 0;
}
```

If any optimizations are enabled (`-O1`, `-O2`, or `-O3`), the compiler will generate code for printing only “Hello,” not for “Goodbye.” Changing the type of `i` to `unsigned` brings back the “Goodbye” generation.

In C++, signed integer overflow is undefined behavior. The compiler doesn't have to assume that `i` will ever wrap around to a negative number or 0. Unsigned integers, in contrast, have well-defined overflow semantics.

Still, you should always default to using debug builds if that's at all possible. If you have to use a release build, pick one for which you have debug information available so you don't have to stare at assembly code (see “Collect Crash Dumps” on [page 91](#)).

Start the Debugging Process

The very first step after building the program is to launch it from the debugger (see “Attach the Debugger by Default” on [page 59](#)). If nothing out of the ordinary happens, your program will run its course. Since the CPU supports debugging in hardware, having a debugger attached doesn't even influence performance in most cases. CPUs support placing breakpoints, single-stepping instructions, and reading register values.

Use Breakpoints

Set up a generous number of breakpoints around the entry point. See “Use Breakpoints to Inspect Your Code’s Execution” on [page 61](#) for an approach. You could also let the program report issues using asserts as programmatic breakpoints (see “Instrument the Program to Report Problems” on [page 62](#)). Once you get to a breakpoint, start stepping through the code and try to discover what your program is doing. As soon as you see symptoms of your bug or a flaw in your implemented logic, make sure that you check all the relevant information the debugger presents.

Understand Conditional Breakpoints

Conditional breakpoints will pause the program only if it hits the breakpoint and the given condition is true. This lets you use them in situations where software reaches a breakpoint often, but only rarely when it’s in the state you’re looking for.

These breakpoints have a few downsides, though. First, the condition-authoring interface makes it tricky to avoid spelling and syntax errors. Second, while the condition is basically code, you often don’t have code completion available in this situation, and you don’t get any feedback about whether the condition is valid. Finally, some platforms implement conditional breakpoints using regular ones, which makes the program pause while the debugger evaluates the condition and decides whether to move forward. This can slow down the debugging process too much for conditional breakpoints to be useful.

It’s better to spell out the `if` condition in the code and put a regular breakpoint in the scope below it. This way you can be sure the breakpoint will reliably trigger, you get the full editing support of your IDE, and the process goes as fast as it does with any other code. However, you must compile the code whenever the condition changes, so you should try to get it right the first time. The implementation of a “fake” conditional breakpoint can look like this:

```
if (my_super_specific_condition) {  
    printf("here\n"); // Put a breakpoint on this line!  
}
```

Once you're done with the condition, don't forget to remove it. It should never be committed to your version control system.

Consider the Call Stack

When the program pauses, you need to figure out what point you're at. The debugger might conveniently highlight the current line in the current file already, but it's always worth taking a look at the call stack. That call stack shows your current function at the top and then the sequence of function calls that took you there.

Consider the following code that—while it doesn't do anything useful—you can compare to the upcoming call stack:

```
void f() {
    // breakpoint here
}
void g() {
    // breakpoint here
}
int main(int, char**) {
    if (true) {
        f();
    } else {
        g();
    }
    return 0;
}
```

The call stack (without optimizations) will look like this:

```
>      Test1.exe!f() Line 10 C++
      Test1.exe!main(int __formal, char * * __formal) Line 20
C++
      [External Code]
```

You can see that, indeed, the program has reached the call to `f()` inside the `main()` function. Thus, you can deduce that the `if` condition must have evaluated to true. Call stacks are important because they show what control-

flow decisions the program made and let you navigate to the other stack frames.

TIP: DEAL WITH CORRUPTED CALL STACKS

Every once in a while, I end up with a corrupted call stack that makes it hard to figure out which function I'm in and how I got there. This tends to happen with stack overflows, which usually occur in the case of infinite recursion or when calling a virtual C++ function on a garbage pointer.

It's essential to closely examine what information the call stack still has to offer. It may contain clues to the function's name or the type of object it was going to be called on. Once you find out more, you'll have to carefully inch towards the offending call site by placing breakpoints before the problem occurs.

Assuming your program pauses near where your bug appears in the code, the call stack gives you an initial idea of how the program reached this particular point.

Examine the Data

Once you know where you are, look at the data that contributed to the program reaching a state where the bug can occur. Data influences software's control-flow decisions and ultimately its logic. It's an important puzzle piece when you're investigating a bug. Here are some categories to consider:

Local variables

These variables are the ones you declare inside a function. The debugger identifies them easily and usually gives you a nice list of their names, types, and values. Local variables exist only for the duration of the scope they're declared in, meaning you can often tell their values by reading the code without the need for a debugger. While function parameters behave just like local variables, you initialize them with outside values, so it's more difficult to tell what they contain without a debugger.

Static variables

Static variables use the same scope as local ones, but they last as long as the program does. Every time you enter a scope, the static variable

will refer to the same data its last execution used. While it's still easy to tell what the variable does, it's much harder to reason about its value, so it's generally best to avoid static variables when possible. The debugger usually displays static and local variables together.

Member variables

Member variables or properties are variables in an object-oriented language associated with a class instance, which is the context for the function's execution. They often appear under `this`. While they have the same lifetime as the instance, making their scope conceptually clear, it can be hard to determine the instance's lifetime. Additionally, any member function can read and modify a private member variable, and any code with access to the instance can access it too. This makes it difficult to trace where changes come from and which parts of the program use the data.

Global variables

Global variables are simply accessible from anywhere in the code. If you know the name of a global variable, you can read it or write whatever value you want to it at any time. You may need to specifically debug a global variable to see its value or the changes it's gone through. Common implementations of the singleton pattern make singletons behave just like global variables, so they come with the same challenges.

In most cases, the debugger doesn't automatically display global variables, since there would be too many if it did. Some debuggers try to guess which ones are relevant, but it's usually a hit-or-miss process. If you know a variable's name, though, you can always tell your debugger to display its value.

Memory blocks

Memory blocks hold data and influence your program's behavior. A pointer indicates a memory block's address, letting you theoretically do anything to the memory it contains. You can write any data you want there. You can reinterpret the data as a different type. You can even use pointer arithmetic to reach other memory areas, no matter how far away they are. Thus, tracking how memory block data changes and

where it comes from is even more difficult than it is for global variables.

Debuggers never automatically display raw memory blocks. Most have a memory view that you can use to see the raw binary data (usually in hex format) at a given address. Alternatively, you can cast an address to a pointer type and let the debugger display that.

When you're examining your program in its paused state, you need to consider all the data the code references. It can tell you either how the bug occurred or how the program's current state evolved to let problems happen.

Check Out Other Threads

The debugger shows you a list of threads the program currently uses. You can look at the call stack for each to see what it's doing. This is helpful if your program uses parallelism to distribute its work across multiple threads. It's not uncommon for concurrency to cause bugs, so it's useful to check whether a running thread could interfere with the code you're debugging.

Step Through Code Line by Line

By controlling your program's execution manually, you can learn how its logic behaves. Study the information the debugger presents carefully to see whether it can tell you more about the bug. If it can't, either let the program run until the next breakpoint or single step through the code.

If you believe the program's current state offers nothing to learn or just doesn't apply to the bug, you should continue. The next breakpoint might provide better insights. It's common to skip a few breakpoints either because you placed them too liberally or because you have to bypass the first couple of iterations in a loop.

NOTE

If a breakpoint is indeed unnecessary, remove it before continuing the execution.

However, if you think you've found your program in a state relevant to the bug you're working on, you should tread more carefully. Keeping an eye on the call stack, the variables' values, and where you are in the code,

step forward line by line. After each step, the debugger updates all the information it displays, and you need to re-evaluate what you see and the best next course of action.

You usually have the choice between three different kinds of steps:

- You can step just to the next line (also called *stepping over*).
- You can *step into* a function. If the next line contains a function call, such an action makes sure you enter the function and can debug it. Otherwise, you proceed directly to the next instruction.
- You can also *step out*, which means that the debugger runs the current function to the end and returns control to the calling function.

Sometimes it's helpful to skip a couple of lines ahead by placing a breakpoint there and continuing the execution or using the debugger's Run to Cursor feature, which achieves the same goal.

Note that the build configuration and optimization options influence how well you can step through the code.

Trace Symptoms to the Root Cause

With luck, debugging the program as described helps you understand how the bug happens. You need to identify the symptoms you can observe right there in the debugger. Does one of the variables have the wrong value? Did execution end up in the wrong location in the code? Did the program cause a crash?

From these symptoms, you can reason backward to find the initial problems. If you're in a particular scope in a certain function (where you shouldn't be), a number of `if` conditions must have evaluated to true. If a given variable holds a certain value it shouldn't, you should be able to figure out where the value was assigned.

If what you see in the debugger gives you an idea of how to fix the problem, you should try to do so (see “Deploy a Quick Fix” on [page 201](#)). Afterward, try to reproduce the issue with the same breakpoints.

Debug with the Assembly View

Because the machine code doesn't always map one-to-one to the lines of code you can step through or put breakpoints on, switching to your debugger's assembly view can be worthwhile. Programming languages can

express quite a lot of logic in one single line that then gets compiled into several assembly instructions. Pinpointing a bug on a particular part of a multi-instruction line can be difficult without looking at the code and having at least a rudimentary understanding of it. In the following example, it can be difficult to say exactly where the crash occurs without diving into the assembly code:

```
void f1() {
    int* ptr = NULL;
    for (int i = 0; i < 100; ++i, *ptr = 20) {printf("%d\n", *ptr);}
}
```

The (unoptimized) assembly code looks like this:

```
f1():
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     qword ptr [rbp - 8], 0
    mov     dword ptr [rbp - 12], 0
.LBB0_1:
    cmp     dword ptr [rbp - 12], 100
    jge     .LBB0_4
    mov     rax, qword ptr [rbp - 8]
    mov     esi, dword ptr [rax]
    movabs  rdi, offset .L.str
    mov     al, 0
    call    printf
    mov     dword ptr [rbp - 16], eax
    mov     eax, dword ptr [rbp - 12]
    add     eax, 1
    mov     dword ptr [rbp - 12], eax
    mov     rcx, qword ptr [rbp - 8]
    mov     dword ptr [rcx], 20
    jmp     .LBB0_1
.LBB0_4:
    add     rsp, 16
    pop     rbp
```

```
        ret
.L.str:
        .asciz  "%d\n"
```

The four lines of C code, one of which contains the bug, map to 22 x86 assembly instructions. Only one instruction (`mov rcx, qword ptr [rbp - 8]`), which dereferences a null pointer, is to blame for the crash.

You don't need a deep understanding of assembly to get the gist of it. Here are the most essential x86 instructions:

- mov** Move data between registers, memory, and immediate values.
- add, sub, mul, div** Perform math operations.
- cmp** Compare two values.
- jmp** Unconditionally jump to a different location in the program.
- je, jge, jne, jz, jnz** Jump based on various conditions.
- call, ret** Call or return from a subroutine.
- push, pop** Push data onto the stack or pop data from it.
- and, or, xor** Perform logical operations.

With these, you should be able to tell what's happening in your assembly code, especially if you have your source code available to compare with it. Of course, other CPU architectures have different instruction sets, but they generally use similar principles and functionality.

Launch the Correct Executable

Sometimes the executable you run to start a program isn't the one making it operate. A *launcher* often performs some minor work before starting the actual program. It may handle tasks like checking for updates, downloading the latest version, deferring control to a shared library, or even just providing immediate visual feedback while a slow program starts. If you're aware that such a setup is possible, you can ensure that the executable you launch through the debugger is the one that stays around and the one you want.

In another complication, command line arguments can drastically alter a program's behavior, and you may never get to see them if another

executable starts the software. Of course, you can provide these parameters when launching your program from the debugger, but first you need a way to understand what they are.

TIP: CONSIDER BUILD SYSTEMS

I frequently run into executables that launch others with complicated command lines. They collect some information first, then call other tools to process it. Build systems also follow this pattern, as with the *make* tool, which calls the compiler after setting some prerequisites and determining what needs to be built.

While you can debug the launcher and piece together the command line from there, it's much easier to run it and observe how it takes its course. Once the second executable is running, you should turn to your operating system and get the relevant information.

Some tools can help you find a running program's executable and discover what arguments the launcher supplied. On Unix-like operating systems, running `ps` does the job. Windows requires some more advanced tools, like Process Explorer (<https://learn.microsoft.com/en-us/sysinternals/downloads/process-explorer>) and System Informer (<https://github.com/winsidersss/systeminformer>).

Use Advanced Debugging Features

Learning all the intricacies of a debugger is a considerable topic just by itself. (Take a look at their manuals and tutorials!) This section gives you an overview of a few useful but lesser-known features of Visual Studio's debugger and an idea of what to look for in others.

Visual Studio's Debugger

The most popular IDE on Windows is Visual Studio. It's used for many projects targeting Windows or platforms you can cross-compile from there, such as Android, embedded platforms, or game consoles. Here is a list of less-well-known features that offer great benefits when debugging:

Attach to a process

In addition to launching a program from the debugger, you can attach to a process that started externally and then proceed just like the

debugger had initiated it. There's even a handy *reattach* feature that will attach to the same software. The Debug menu contains both options.

Symbols

If you write software yourself, you'll have its symbols available, but you may have to explicitly point Visual Studio at them. The Modules window will let you do this, or you can find it under Tools ► Options ► Debugging ► Symbols.

Execution pointer

It's possible not only to step forward through the code but also to move the execution pointer around freely, repeating or jumping over lines.

Executing some code again is useful in case you weren't paying enough attention the first time or you set your breakpoint too far down. Jumping over a bit of code helps when you know there is a problem with it, but you still want to see what subsequent code does. While it's also possible to do this with code changes and relaunching the program, using the debugger is often faster.

It's not a good idea to skip around in your code too much, though, because you can miss fragments that other pieces require and probably make everything fall apart. Also, be careful because moving the pointer isn't actual reverse debugging. If you drag it back up, the debugger doesn't undo any effects the already executed code caused.

Variable visualization

A Visual Studio feature that lets you define how the debugger visualizes instances of user-defined types. It works by providing a Natvis file containing an XML description of the visualization (<https://learn.microsoft.com/en-us/visualstudio/debugger/create-custom-views-of-native-objects>), helping you perform tasks like defining custom container types just as if they were part of the standard library. You don't have to fiddle with raw pointers and pointer arithmetic to display a certain element in your custom vector implementation.

Watch window

Visual Studio's Watch window, which you can find under Debug ► Windows ► Watch ► Watch 1, is a powerful tool as well. In addition

to displaying variable values, it lets you enter complex expressions involving operators for tasks like calculations, casting, and pointer arithmetic. You can even call functions for the debugger to execute while the program is paused. (Function calls don't work in all cases, but they're very powerful when they do.) If you want to just run your expression once, you can do all these things in the Immediate window. The Watch window also allows you to add little format specifiers to your expression, such as putting `,10` behind the expression to see an array with 10 elements. This even works for pointers to primitive types, which Natvis doesn't support.

Selectively disable optimization

Sometimes you have to debug release builds because a bug only manifests there, but the optimized code makes debugging harder than it should be. It may not be possible to disable optimizations for the whole program because of performance reasons or because that might make the bug disappear. However, you can disable optimizations for arbitrary sections of a source file in Visual Studio. This is how it works:

```
#pragma optimize("", off)
// unoptimized code is here
#pragma optimize("", on)
```

Data breakpoints

Data breakpoints make the debugger pause execution whenever the data at a specific memory address changes. You have to set them up anew every time you run your program because the memory addresses of your variables change between each execution. This feature is especially helpful when a variable is corrupted and you just can't determine what part of your code is responsible for writing to it.

Under Debug, select **New Breakpoint**, then **Data Breakpoint**. You can either provide a memory address directly or specify an expression that evaluates to an address. The size of the memory area to watch depends on the target platform, but it tends to be very limited (1 to 8 bytes on 64-bit x86).

IN PRACTICE: ATTACH AFTER THE BUG OCCURS

A program I once worked on had a bug that wouldn't occur with the debugger. I believe that this was due to timing issues. As this software was running on a remote device, having the debugger attached forced it to send its log to my development machine and communicate about its current execution state.

In this case, I started the program on its own, reproduced the issue, and only then attached the debugger. While I couldn't observe the bug happening, I could still examine its fallout. Having the information about what the bug did to the runtime data eventually allowed me to fix it.

Other Debuggers

In addition to Visual Studio, GDB and LLDB handle native code for languages like C, C++, D, Go, Objective-C, Rust, and Swift. Your target platform and operating system determine what you use: for instance, GDB is common for debugging in Linux and targeting embedded devices. For code running in virtual machines, various specialized debuggers work with Java, Lua, Perl, Python, Ruby, and so on. Whenever you're faced with a new language, you should spend time finding the right debugger for you and learning about its features.

14. Read the Code

How? Go through the code meticulously and expect anything to break. Consider under what circumstances it might expose your bug.

Why? Reading the code allows you to see the bigger picture and ignore irrelevant details. Check out your surroundings, look at other systems, and follow up on how something *might* break. This strategy brings your human intuition into the debugging process.

Reading the code also helps you understand your program's implementation, what makes it work, and its potential issues. With some experience, you can identify weak points or suspicious areas that you can check out first when investigating bugs (see "Try Shortcuts" on [page 38](#)).

To understand the code and identify problems in it, you must thoroughly understand the programming language you're using and the program's basic functions. It's also helpful to understand any APIs you use.

Act Like a Virtual Machine

Reading code to find a bug is a bit like executing it in your head. You become the virtual machine. To do that well, you have to understand the programming language thoroughly; if you don't grasp its intricacies, you'll overlook some of the code's potential effects.

Walk through the code one line at a time. If a line is an assignment, remember the value. If it's a function call, go into that function. If it's a calculation, do it yourself, or at least estimate the result. You can lean on intuition to skip over statements you're confident in (boilerplate generally works the same way every time). While routine makes it possible to introduce bugs into boilerplate, the programming language hopefully makes it hard to do so and easy to spot issues.

For example, look at this:

```
int sum = 0;
for (int i = 0; i < 100; ++i) {
    sum += i;
}
```

You can easily derive that this code will run 100 iterations and none of them will cause a bug. If you need `sum` for further calculations, you should find an easier way to get the correct value than making 100 additions in your head.

In other cases, it may be obvious that one particular iteration will cause a bug:

```
int arr[] = {0, 1, 2, 3};
int* ptr[] = {arr+0, arr+1, 0, arr+3};
int sum = 0;
for (int i = 0; i < 4; ++i) {
    sum += *ptr[i];
}
```

By following the code closely, you can tell that `i == 2` will result in a null-pointer access and the program will crash.

Find the right balance between evaluating every single instruction and glancing over bigger sections. Start by skimming the code and looking for obvious problems. If you can't find anything relevant, begin paying closer attention and validating every single instruction. Don't exclude code paths or functions because you presume they can't fail. The bug may hide in an unexpected corner (see "Question Everything" on [page 127](#)).

When running the code through your "brain VM," you can start from either the program's actual entry point or from an area in the code you've already identified as related to your bug. You should choose the main entry point (the `main()` function) in cases where you don't know how the bug comes into existence. When you know the bug's location, focus initially on its surroundings. The process gets more complicated as the number of components increases. Having just one program or executable to consider means you have only one language and codebase to dig into. However, many software packages combine multiple programs with databases, network connections, or APIs. To follow one execution from start to finish means you have to switch programs and potentially languages at least once.

IN PRACTICE: WATCH OUT FOR MULTIPLE LANGUAGES INVOLVED IN ONE BUG

Sometimes I deal with HTTP requests, most commonly when a client written in C or C++ needs some data from one of our server-side applications, which usually aren't in those languages. Whenever I run into an issue, I have to debug both the frontend in C/C++ and the backend in Java, JavaScript, or Python.

A similar situation arose when a game and its engine were in C++ while the GUI was exclusively implemented in Lua. While one process contained all of them, I still had to go back and forth between the different languages and codebases whenever a bug occurred in the GUI. If your program interacts with another, you have two options. The hard way involves following along and reading the other code too. Since that means you have to switch contexts, ideally take notes so you know where you were when you left off.

The alternative is to *mock* the other program and assume it returns a specific result, which means you don't have to debug it. However, if your

program's behavior makes the other one seem like the potential culprit, you'll need to read its code after all.

You should base your mock result on real-world observations, since you want to investigate a use case that can actually happen and that reproduces your bug. If you can't find an appropriate result, you'll have to dive into the code paths your software chooses depending on the other program's return value. Choose a value that's likely to reproduce the issue.

Strive for Readable Code

On the flip side, the easier code is to read, the easier it is to mentally execute and the less likely it is to hide a bug from you in the first place. You usually can't control how readable the code in front of you is, but understanding what makes code readable sharpens your eye for the spots where unreadable code tends to conceal problems.

What counts as readable depends a lot on the language and coding conventions you're working with, and it's partly subjective. Still, some widely accepted standards make code easier to follow:

- Use consistent formatting (indentation, spacing, naming conventions, and so on).
- Choose descriptive and meaningful names for variables, functions, classes, files, folders, and other elements.
- Follow established best practices and patterns.
- Break your code into logical modules, functions, or classes that perform specific tasks. Each should have a clear purpose and be self-contained.
- Keep your code concise and avoid duplicating logic or introducing unnecessary dependencies.
- Use named constants or variables instead of magic numbers and hardcoded values.
- Implement proper error handling and provide meaningful error messages (see “Instrument the Program to Report Problems” on [page 62](#)).
- Add comments and provide documentation explaining your code's purpose, logic, usage, assumptions, and potential pitfalls. However,

keep it concise and ensure your code is self-explanatory wherever possible.

Follow two general principles to make your code readable:

Avoid surprises

Anyone should feel right at home in the code you write. That's relatively easy to achieve if you ensure it looks and works like most other code in the project. Also, try to make your code as simple as possible. Use algorithms and abstractions that are just about complex enough to fulfill the system's implementation requirements.

Choose descriptive names

Names should tell the reader what they need to know to decide whether they should dive into a function definition. Avoid single-letter or ambiguous names that make it difficult to understand the purpose of a particular element. Well-established conventions, such as single-letter loop counters, are an acceptable exception.

The original author will always find it easy to read their code right after they've written it, but readability diminishes drastically with time.

Successful teams implement regular code reviews.

Some languages are more verbose than others, which also has an impact on readability. More concise language could offer increased brevity and therefore a decreased cognitive load, while more verbose language can provide complete understanding because it explains the code in greater detail.

Additionally, a statically typed language always makes variable types clear, whereas dynamic typing is easier to write but more difficult to read.

For example, this piece of code shows a common pattern in C++:

```
std::vector<std::shared_ptr<int*>> data;
--snip--
for (auto it = data.begin(); it != data.end(); ++i) {
    auto element = *it;
}
```

While C++ is statically typed, the `auto` keyword allows limited dynamic typing. Of course, `it` will be some kind of iterator, but it's hard to

say exactly what it contains. How often do you have to dereference the iterator to get to the `int` value?

IDEs can assist in many such situations, but sometimes even they don't know the exact type you're dealing with, and they're not available in all cases. Working with code review tools, examining differences in your version control system, or making minor changes outside your main development machine will often block access to an IDE.

Always prioritize writing readable code. You'll probably just write it once, but people will read it much more often over its lifetime, which might last for decades. If it's hard to grasp, it'll probably lead to bugs down the line.

Record What You Find

As you read through the code and work out what it actually does, keep a record of what you learn. Note the value each variable holds at a given point, the result of a calculation, and the path execution takes. Writing it down keeps you from re-deriving the same thing every time you revisit a section, and it lets patterns surface that you'd miss holding it all in your head.

Write down any intermediate results you observe in the debugger or determine by other means so you can avoid repeating your work when reading the same section or considering code that the program often calls into. Make sure to take varying input parameters into account. If you log those results in an organized way, you may be able to identify even small issues when they form patterns.

You can log intermediate results in any text editor or just on a piece of paper. With more in-depth analyses, it's helpful to organize the input values and their outputs in a spreadsheet. That also lets you create charts and graphs from those values in order to identify problems.

To calculate intermediate results, you can also turn to external tools. A basic calculator can confirm whether a formula yields the expected result. More advanced tools include graphing calculators or even software like Mathematica (<https://www.wolfram.com/mathematica/>), which offers a wide range of tools for technical computing.

Know Your Programming Languages

Reading code requires you to know the language it's written in. A lot of popular languages share many common syntax aspects, so you can probably read most of the code out there. However, some are very different from the ones you know. Little pitfalls and discrepancies are less obvious than major differences, so they more easily lead to faulty assumptions and therefore bugs. That's why it's important to know the language of the code you're reading in detail.

Be suspicious of language features that put code's readability at risk. A language may have many different ways to trigger functionality without the code obviously calling it. You must follow the code as closely as possible and keep the relevant features in mind.

Always keep the language reference handy. While it's great to know all the intricacies yourself, that might be impossible for certain languages. It can also be sufficient to know about potential corner cases so you can look up the details when you encounter a suspicious piece of code.

Common Pitfalls

Since many programming languages share roots and inspirations, they also have common pitfalls.

Implicit type conversions

Implicit type conversions are a prime example of an operation that happens without being obvious. They occur when code converts a certain type of value to a different type that can't completely represent the original. For example, converting an integer to a floating-point number isn't always lossless. The other way around loses even more details.

Languages like C++ even allow you to specify and implement implicit conversions yourself. Explicit conversions can exhibit the same problematic behavior, though it's more evident because you specify the target type yourself. In general, watch out for conversions where data can be lost.

Exceptions

Exceptions lead to unclear control flows. As they don't have clear exit points and can unwind multiple stack frames, it's complicated to tell where they come from and how they affect the code's logic.

Polymorphism

Polymorphism is another feature of object-oriented languages that can produce unclear control flow. You might know the type of a certain instance, but not necessarily of the implementation it provides, which can get messy if you call a function on it. This is even more of a problem if the language supports multiple inheritance, as that leads to ambiguities and unexpected behavior even more easily. Many teams disallow the use of multiple inheritance for this reason.

Operator overloading

Operator overloading means you can provide custom implementations for operators such as `+`, `-`, or even `->` for user-defined types. You may expect that adding two variables of a user-defined type with the `+` operator yields the sum of the two, but in reality, something completely unrelated may happen, like the program opening a new window and asking the user for the result. This is an example where the problems would be obvious, but little mistakes in such foundational features can go unnoticed for a long time because no one would ever expect an operator to misbehave.

C++

Several features and corner cases in C++ specifically complicate reading the code. This section lists a few examples you should look out for.

Constructors and destructors

A constructor starts the lifetime of a user-defined type's object, and a destructor ends it. Since these two functions aren't called explicitly, it's easy to overlook them when reading the code and, for example, not stepping into their implementations. Make sure to keep them in mind, especially when it comes to bugs related to an object's lifetime and initial values.

Lifetime

C++ has complex rules regarding an object's lifetime and scope. It's crucial to understand what the scope of a given instance is. Temporary objects are especially problematic as they're never assigned a name, and it's often unclear when exactly they get cleaned up. For example,

you can pass them to a function via a const reference but not any other kind.

Pointers and references

Pointers and references can lead to many different bugs. Most people know they're risky, so they pay close attention when reading code they're in. Still, because they allow you to do almost everything, they easily create memory leaks, invalid memory access, dangling pointers, crashes, data corruption, or undefined behavior.

Strings

String comparisons are also a potential source of issues. Due to C's heritage, char-pointers, that is, `char*`, are common in C++ as well. However, when comparing a `char*` with a string literal, you'll compare the addresses where the data is stored and not the string contents. You have to use instances of `std::string` or the `strcmp()` function instead. Even worse, strings can be interned, so the address comparison may actually work in some cases but not in others.

Passing arrays

One more potential issue related to pointers and arrays is that you can't pass arrays to a function in C or C++. The fixed-size array declaration in a function signature always decays to a pointer type.

IN PRACTICE: WATCH OUT FOR QUESTIONABLE LANGUAGE FEATURES

I once worked on a tool that precalculated how far away from the camera a given object should become visible. It had a function like this:

```
void store_offsets(float distance_offsets[4]) {  
    for (int i = 0; i < sizeof(distance_offsets)/4; ++i) {  
        // ...  
    }  
}
```

It was supposed to store four floats in some kind of data structure. However, the way the function is implemented, it will only ever store one float. This problem led to many objects missing whenever they were supposed to be displayed at runtime. Tracking down this bug in the tools took several people multiple days.

Undefined behavior

The language allows certain constructs that result in undefined behavior, such as accessing an array out of bounds, using uninitialized variables, or performing division by zero. While none of these has a clearly defined erroneous outcome, undefined behavior can affect the rest of the program in unexpected ways.

Macros

C++ supports macros: preprocessor directives that perform text substitution. Macros can be error-prone because they aren't type-safe and can result in unexpected behavior due to incorrect expansion. For example, a macro can expand a passed parameter multiple times.

The following code will print `Result: 2` because the increment is executed both in the comparison and during the `printf()` call:

```
#define TEST(x) if (x > 0) printf("Result: %d\n", x);  
int i = 0;  
TEST(++i);
```

Macros can even overwrite existing names without it being obvious in the rest of the code. Someone could `#define if(x) if(! (x))`, which would compile and mess up most of your logic. This is an exaggerated example, but it's clear that you must carefully consider what macros your code contains when you compile it.

Debugging macros is difficult. Their substitution happens as part of the build process, where you can't interfere, and they change the code without leaving any evidence in your source file.

Variable hiding and shadowing

One last not-so-obvious potential issue is variable hiding and shadowing. Declaring a local variable with the same name as a global variable will not cause an error, nor will declaring one with the same name as a variable in an enclosing scope, but it can cause significant confusion. While compilers are getting good at warning about this particular type of problem, yours may not have the warning enabled for

your project, or you might simply miss it among the rest of the compiler's output.

C#

C# may be higher-level than C++, but that means it has more abstractions that can hide issues or surprise you.

Properties

The language features properties, mechanisms for encapsulating fields within a class and providing controlled access to their values. As you can implement the getters and setters using arbitrary code, simply assigning a value to a variable or reading one can very well have unrelated consequences.

Value types and reference types

Value types (structs) and reference types (classes) behave differently, but you can define both. Understanding the differences between these types and their behavior when passed as arguments, assigned to variables, or used in comparisons is crucial. For example, value types compare the actual values, while reference types compare the object references.

Boxing can turn value types into reference types, and unboxing will do the opposite. A boxed integer behaves very differently from a regular one with the same impact on comparisons. For example, when using equals signs (`==`), they'll compare references rather than their contents.

Garbage collection

While C# runs automatic memory management via garbage collection, you still have to worry about it in some cases. For instance, you'll risk data loss and other unintended effects if you don't clean up unmanaged resources like file handles, network links, or database connections when you're done with them.

String comparison

The language provides different methods for comparing strings, such as `==`, `Equals()`, and `String.Compare()`. The wrong comparison method

can lead to incorrect results, especially when dealing with non-ASCII characters or culture-specific comparisons.

Python

Even Python has its share of relatively unknown corner cases and pitfalls. The following features can sometimes lead to unexpected behavior and bugs:

Default arguments

When using mutable objects like lists or dictionaries as default arguments in function definitions (like `def append_to_list(value, my_list=[])`), changes made to the object persist across multiple function calls. Unexpected behavior can result if the default argument is modified and you expect it to be a unique instance every time the function is called.

Local variables

Variables defined inside a nested scope are accessible outside that scope, even after its end. This behavior can be surprising if the intention is to limit the variable's lifetime. You can define a nested function to restrict a variable's scope.

Inner functions can access variables from the enclosing scope when defining nested functions. If you try to modify a variable from the outer scope within the inner function, though, Python creates a new local variable instead. To modify outer variables, you need to use the `nonlocal` keyword.

Closures

Closures in Python have a feature called late binding. If a variable used in a closure is defined in the enclosing scope and modified afterward, the closure will use the updated value rather than the one it had at creation.

Modifying types

Python allows you to add attributes to objects dynamically, even to built-in types and modules. While this can be powerful, it can also lead to bugs if you accidentally modify an object's namespace or override

existing attributes. Anyone using your code most likely won't expect built-in types to have been modified.

Python 2 vs. Python 3

Furthermore, Python 2.x behaves differently from Python 3.x in some ways. For example, the division operator `/` between two integers performs floor division in 2.x, while in Python 3.x, it performs true division. If Python 2.x executes code someone wrote with the expectation of true division, subtle bugs can result.

15. Follow the Data

How? Follow the data your program uses and transforms to find out where it first becomes bad. You can also identify problematic sections or patterns by comparing good and bad data.

Why? Following the data through its transformations can teach you how your program works, and tracing bad data back to where it became bad can lead you to your bug's root cause. You're focusing less on the logic flow but instead on how the data makes its way through the program and how it gets transformed.

Data, in this case, is a very generic term that covers several different concepts:

- Data is the information your program works on. In most software, the user provides some input data, and the program produces output data based on that.
- Data are the assets that your program needs to function. These include layout descriptions, images and textures, and audio files.
- Data are the settings that control your program's behavior.
- Data are transient or temporary results that your program generates while fulfilling its intended function.

The reality about data is that it rarely just exists. As soon as a program is involved, data is *transformed*. Your software takes the data, changes it, and only then passes it on to the user or some other program. This

transformation can take many different forms as well. Here are some examples:

- Transformation changes the data's representation without changing the information it contains, as happens in lossless compression or converting from one format to another that supports the same feature set.
- Transformation modifies the data, but in a way that lets it serve the same purpose, like lossy compression or a format that supports only a subset of the initial features.
- Transformation produces new information from one or more inputs, such as a service bringing several databases together to provide a useful tool.

Bugs frequently show up in transformation, as that's what you implement when creating your program in the first place. That's why this strategy focuses on following the data through its transformations to determine which one causes your bug.

Find the Original Data

Original input data can come from a file, code, memory, or one of many other potential sources. Access to this pristine data will let you compare it to the output data and determine whether you got the correct result. It also gives you a place to start your investigation.

The data's source determines how easy it is to find the original input. If you're getting it from the user, you can specify exactly what you want and where it should be and can also easily analyze the information with an appropriate tool.

When dealing with data the program gets from elsewhere without user instruction, tracking down the original input is harder. You can employ strategies such as analyzing the source code (see "Read the Code" on [page 113](#)) or debugging it (see "Step Through the Code" on [page 100](#)). Another way to find input data is to determine which files your program accesses, using specialized tools (see "Use Off-the-Shelf Tools" on [page 164](#)). Knowing what the original data looks like helps, even if you have only a partial string or fragment (see "Search in All Files" on [page 86](#)).

Spot the Symptoms of Bad Data

Bad data doesn't fulfill its intended purpose. It could just fail to provide the results the user wants, or it could be corrupted and therefore unusable. You need to fix both cases, of course, but they happen for different reasons.

Look Out for Undesired Data

If the data doesn't provide the information the user is looking for, something is wrong with the program's logic. The specification or translation into code could be wrong. It can be hard to find mistakes in logic, as they might be very subtle or only apply to a small subset of input data. You have to identify under what circumstances the data is wrong, which you can do by gathering user reports or asking your QA team to provide inputs that cause problems.

Identify Corrupt Data

On the other hand, corrupt data always comes from an implementation bug where a transformation turns good input data bad. The reason is usually a standard programming mistake or a misinterpretation of the implemented algorithm.

Identifying corrupt data can be difficult too, but it doesn't always have to be. Corrupt data often looks like it doesn't belong or follows specific patterns that give away its nature. Look for any data that doesn't align with your expectations for its type, such as a negative number for a file size or the elements in an array. Huge and floating-point numbers should also make you suspicious. Also, look out for special values in floating-point computations as specified by the IEEE 754 standard:

- Positive infinity indicates overflow or division by zero.
- Negative infinity indicates overflow or division by zero with a negative result.
- Not a Number (NaN) is a special value that represents the result of an undefined or indeterminate operation, such as 0 divided by 0. It's commonly used to indicate errors or undefined results.
- Denormalized numbers, or subnormal numbers, are floating-point values smaller than the smallest normalized number. They're used to

represent values that aren't large enough for the normal floating-point format.

These special float values play important roles in representing exceptional conditions and handling edge cases. Whenever you encounter them, you should be exceptionally suspicious. NaN values even propagate through any calculations they're part of. That means most operations involving a NaN will yield a NaN as a result independent of what other value is provided as an input.

Text or strings are also easy to check for bad data. If they contain garbled text you can't read, you should first make sure you're using the correct character encoding. If the encoding is correct and the text is still garbled, the data is obviously corrupt.

It's a bit trickier to spot corrupt data if the text could be in a foreign language or use a foreign character set. Online translation tools can help, as can asking someone who speaks the language in question.

Strings you encounter may also contain human-readable structured data in a format like JSON, XML, or YAML. Strict requirements about the structure for such data make it possible to detect even minor corruption in a string like that. If there's a quotation mark or angle bracket in the wrong place or the data doesn't start with the correct prefix, you can be suspicious.

Pointers are data as well: a few bytes that store an address. Their size depends on the bit-width of your program's architecture. For a 32-bit architecture, the pointer size is 4 bytes, and it's 8 bytes in a 64-bit architecture.

Be immediately suspicious of null pointers. They may be called `nil`, `0`, `NULL`, or `nullptr`, depending on your programming language. Whenever you encounter them, you should double-check whether having a null pointer is valid and whether your program reacts appropriately to its presence.

As pointers have to store valid addresses in memory, you can make certain assumptions about them. The address is commonly aligned, so you can access an instance of the type the pointer describes there. Usually, this means an alignment of at least 4 bytes, which means the address should be divisible by 4. The address should be reasonable in relation to any other pointers your program deals with. If it points to a memory location close to

0x0 or the upper end of the addressable memory, you need to take some time to investigate.

IN PRACTICE: PAY CLOSE ATTENTION TO YOUR DATA

I once worked on a C++ program that had been ported from a 32-bit architecture to 64-bit. This program was written in a very low-level style and dealt a lot with raw pointers. Most of it worked after the conversion, but we had a hard time investigating some minor issues.

Then we realized that all pointers' upper 4 bytes were full of zeroes (for example, 0x00000000ABAAFE30) and suspected that they were missing some address data. Sure enough, the pointers were stored in 32-bit `uint32` variables before being converted back to actual pointers. This just cut off the upper half of the pointers and corrupted many of them.

Observe Bad Data

Now that you know what bad data looks like, you should observe it in action, and you have several options for doing so. The most flexible method that doesn't change the program is to use your debugger's variables or memory view, depending on what you're looking for (see "Step Through the Code" on [page 100](#)). This lets you poke around the data and view the memory at arbitrary addresses.

Another option is viewing the data as the program presents it to you. You can see it on the screen displayed in the GUI or rendered images, and you may be able to observe it in the log. Also, if your program generates output data, you should be able to view that either in the files the software writes or as the data it sends to other programs or services.

In some unfortunate cases, your program may not expose the data you're looking for by default, so you may have to tweak it to get a look. It's best to change the configuration rather than the code, if possible, because that has less chance of making the bug less reproducible and doesn't require you to go through the build process again.

Whether the data is permanent or transient chiefly determines how easily you can examine it. Programs store permanent data to load later, generally in a displayable form, but transient data isn't always easy to present. It might be optimized for another use case, such as storage size,

communication with another program, or algorithmic processing. You may need to implement a conversion function, being very careful not to accidentally fix your bug or add any new ones that would keep you from identifying the original problem.

Some data formats are impossible to view in the debugger. For example, seeing raw bytes from images, 3D meshes, audio samples, or videos doesn't tell you anything. You need dedicated tools to really examine such data, which means you'll need to write it to a file one of the tools in question can open or copy the raw bytes from the debugger's memory into a file you can view.

Find Where the Data Becomes Bad

To find the cause of the bug you're working on, pinpoint where the data goes wrong. That spot might be the bug itself or a lead you can track down. You can work either backward from the flawed input or forward from a state where the data's still good to find the problem, and both using a debugger or running the program can help.

Track Data Changes with a Debugger

Using the debugger to find bad data means you directly deal with your program's code. The code is what implements the data transformations, and it's most likely to blame for the bug.

As a first step, follow the transformations by reading the code and note all the spots where it modifies the data. (You can also do this using the debugger and stepping through the code.) One way to track all the transformations is to place breakpoints wherever the data changes. When the program hits a breakpoint, check the state of the data. Continue if it's still good. If not, go back and add breakpoints in between when it was last good and when it became bad. This way, you narrow down the portion of the code where the bug can appear.

Data breakpoints have their limitations, but if they work in your case, they're incredible tools. Your data can't exceed the maximum size of a breakpoint (usually up to 8 bytes), but that's something you can work around by just looking for changes in a small portion of a bigger memory range that goes bad.

A data breakpoint just watches one particular memory address, though, so your data has to reside in one place. That means breakpoints work best for global variables, which always stay where they were created, and for memory allocations on the heap, where only the pointer indicating them moves. Breakpoints also pause the program every time it hits them, which can be annoying and time-consuming in cases where you have to observe the data over longer periods. One alternative is displaying the data on the log output every time it's modified. There, you can check if it's still good and get an idea of how it evolves over time, when the issue occurs, and what the bad data looks like (see “Debug with `printf`” on [page 158](#)).

In addition to logging your data, you can also add code that verifies it by comparing it against data you know to be good. Use asserts or similar tools and alert yourself whenever a problem occurs (see “Instrument the Program to Report Problems” on [page 62](#)). This approach scales very well for long-running programs and rare cases of bad data.

Reproduce the Bug in the Program

If you're investigating visible data, you may be able to simply view it in your program and observe when it goes bad. As soon as that happens, take note of the action you just performed, because you'll have to investigate it as a next step.

Some tools automatically instrument your program to report data corruption (see “Use Off-the-Shelf Tools” on [page 164](#)).

Compare Good and Bad Data

Comparing good and bad data may reveal a pattern or systematic difference. For example, UTF-8, UTF-16, and UTF-32 are all different character encodings, but if the API's expectations are unclear, it can easily pass around the wrong one. If a program expects a UTF-8 string and gets a UTF-16, the result is a UTF-8 with `0x00000000` for every second character, which is pretty easy to figure out once you know it can happen.

The comparison can also tell you when the data appears in an unexpected format. For example, it's easy to see if you get XML instead of JSON, or text instead of binary data. Intuitively spotting situations like compression problems when all data is in the right format is a lot harder, though.

You can use a diff tool to compare text data, find comparison tools and algorithms for images, and probably track down similar software for other common formats. For custom formats, you'll need to decide whether comparison is worth making custom tools or it's better to try another investigation strategy.

Follow Good Data

It can also be useful to follow good data on its way through your program, as doing so will help you understand how the different pieces work together. If your bug doesn't manifest in the data itself, but you know about related areas, following good data there can show you more about the bug's connections.

16. Question Everything

How? Verify all your assumptions. Even if you don't expect a particular system to contain your bug's root cause, thoroughly analyze the associated sources. Also, examine the information your program takes in and the environment where it runs.

Why? Nobody's perfect, and your bug's root cause must be hiding somewhere. If the cause isn't apparent or is challenging to find, it's likely hiding in an unexpected area, which you may have missed because of your assumptions. Even if you can't find the cause with this method, diving into the details of your program gives you valuable background information.

This strategy is about throwing all your assumptions and biases overboard and instead putting everything up for discussion and examination. You may have to dig into unfamiliar code, other kinds of sources, your program's environment, or third-party software. While none of this is easy, it becomes mandatory once you've exhausted all the more straightforward options.

Find the Cause That Must Exist

It makes a lot of sense to look into seemingly unrelated areas once you don't have any better lines of inquiry. A cause for your bug must exist somewhere in your program or the environment it runs in.

Start questioning everything about your program. Examine its sources, its environment, the inputs, and all its other components. If that fails, you may have to perform an exhaustive search. Disassemble your program into the smallest pieces and check all of them until you find the one responsible for the bug. You can apply the *duck test*: If it looks like a duck, swims like a duck, and quacks like a duck, then it's probably a duck. You could say a piece that behaves erroneously under specific inputs doesn't contain a bug and the inputs are to blame. However, ideally every single piece should be designed and built so that it's resilient against such issues.

Of course, exhaustively searching all potential hiding places for your bug takes way too long in any somewhat complex program. You need intuition, experience, and luck so you can identify the right places to check (see "Try Shortcuts" on [page 38](#)).

Don't Trust Blindly

Everyone makes mistakes, and those mistakes can go unnoticed for a long time. You may be hesitant to distrust some software or a piece of code because someone else created it or it's been in use for a while. However, while these things make bugs unlikely, they don't guarantee that one will never show up.

Moreover, a bug's cause and effect aren't necessarily closely related. It's not uncommon for a bug's symptoms to show up in a completely different part of the program from the one where it's located. Making this connection isn't always easy or feasible, and sometimes it's downright impossible. That's why, especially in cases where the connection isn't obvious, you must use your experience and intuition to investigate seemingly irrelevant areas.

IN PRACTICE: WATCH OUT FOR SURPRISING CONNECTIONS

In a game I worked on for a while, a crash showed up after release. Because we had crash dumps implemented for this project, we knew it failed somewhere in the third-party physics simulation. The bug's reproduction was much more difficult because it was hard to tell how the game could crash there.

A colleague observed that the game would always crash when the player jumped on a particular object. This object showed up in a lot of different areas, and therefore the crash would also happen all over the place. The connection was unexpected and

not something anyone on the team would have usually investigated. Ultimately, it turned out that the object's collision shape hadn't been set up correctly.

Even code that has worked perfectly for years can suddenly produce a bug. The bug may not be new, though; it may have been there all along and surfaced only when something outside the code changed.

Investigate Assumptions and Dependencies

When in doubt, drill into all the different areas that make up your program and its environment. Each of them may contain the bug itself or may trigger it.

Don't Skip Trusted Code

Don't stop digging and debugging just because you get to the point where your program calls into code that you're not familiar with. Third-party products, for instance, can have bugs as well. They're often written under specific expectations that you may not know about or that your program may not be compatible with.

Even if the code doesn't contain any bugs, looking into it may still give you an idea about its expectations and requirements. You may just be passing the data in the wrong format, or you may be calling a given function in the wrong order or at the wrong moment in time. If you dive into what the code is doing with your data or what expectations it has, you can figure out what you should be doing differently.

TIP: BE CAREFUL NOT TO GET LOST IN IRRELEVANT DETAILS

Don't go so deep into analysis that you get lost in irrelevant details. Always keep an eye on the goal and follow up only on meaningful leads. It can certainly be useful to get a more comprehensive understanding of your program in all its glorious details, but you have to prioritize that work so that you can fix your bug in a reasonable amount of time. See "Keep a Debugging Log" on [page 140](#) for the best way to do this.

You should even dive into code that the industry views as the immutable foundation of all the rest. Even your programming language's standard library, drivers, and graphics APIs can potentially contain bugs.

They're undoubtedly well-tested and used by many people and programs, but they're by no means perfect. Looking at changelogs of new releases proves that.

The same goes for data structures. They may seem flawless, but that's rarely the case. Both structures that are part of the standard library and those other libraries provide can contain issues. When it seems like a bug is associated with a data structure, feel free to get in there and debug it. You might not be familiar with the structure's internal organization, but you may still find helpful bits and pieces to help you understand why a particular bug is happening. Alternatively, replace it with a simpler one that's less likely to be broken. If it works, you may be able to just use that structure without finding the actual bug or compare the two structures' behavior to identify the problem.

TIP: LOOK INTO THE INTERNALS OF STANDARD LIBRARY TYPES

Mutex implementations often internally store the ID of the thread holding the lock, and using the debugger to peek into the values inside a mutex's data can be useful when dealing with deadlocks. I usually can see which thread is getting in the way just by checking the mutex for the one waiting to acquire a lock. Once I know that and I can find the relevant data, figuring out why a deadlock occurs becomes very straightforward.

Hardware abstraction layers, which sit between the hardware and your code, are commonly called drivers if the operating system manages them and a unified API offers them. Users rarely have to deal with drivers anymore: They're rather standardized, and the system generally installs them as soon as you plug in a new piece of hardware. However, they're still programs that can have their own bugs.

Dig into known issues for the drivers your program has to interface with. If they're open source, and almost all drivers for open source systems like Linux are, you can check out the code and potentially debug it. Sometimes the source code may be difficult to find, though.

Graphics APIs like Direct3D, Metal, OpenGL, and Vulkan present a similar case. They may seem like an immutable layer between the graphics

processing unit (GPU) and your program, but they're common APIs that the graphics drivers implement.

IN PRACTICE: BE SUSPICIOUS OF NEW DRIVERS

When we were working on implementing a Vulkan renderer for Android devices, it was very early, and the first devices had just started shipping with this graphics API. In the process of getting our renderer up and running, we kept bumping into problems.

We tried to figure out the problems using the debugger, which didn't help because we didn't have access to any source code behind the Vulkan API calls. Comparing our implementation to the official Vulkan API specification and the few samples we could find on the internet didn't help either. In the end, we concluded that the drivers implementing Vulkan were buggy on several devices. That was the first time I lost some trust in drivers and when I started looking a lot closer into issues where such basic layers may be to blame.

You may be operating a lot closer to the hardware when working on embedded devices. In these cases, you may have to investigate the drivers more closely (if they exist) or start debugging the firmware.

Finally, you may be reluctant to debug code internal to your company that it has used in production for a long time. You or your respected colleagues may have written this code. It may be known as battle-proven and have an air of being infallible. After all, if it survived this long and hasn't been blamed for any bugs, how could it have any issues? However, a change in the environment or a different use case could cause old code to fail. For example, your newly written code might trigger an edge case that has remained hidden for years.

Investigate External Data

Data read from files and other programming languages can lead to bugs as well, especially because they're different from your usual code, and some files can contain vital assets or configurations. When you come to a file or a language switch, note where you are in the code, then investigate the external data and the code's interaction with it.

Check for Unexpected Use Cases

Testers and users may ask your program to do something you didn't design it for. This might be fine in a lot of cases, but in others it can cause problems, even in software that has worked without issues for a long time. For example, unintended use may push the input an order of magnitude past what the system can handle, make the processing run so long it seems to never finish, or overflow a fixed-size buffer so that the program runs out of memory.

Look for Bugs Related to External Interactions

Interacting with external entities (like users, other programs, or even the physical world) is always a risk because anything external can change independently of your program.

You should always investigate user interaction when you find a bug. Users might enter unexpected information, format it differently, skip required steps, or complete them out of order. While user action often takes place through classic interfaces like mice, keyboards, and screens, it can also involve less common devices, like a sensor that counts people as they enter a room.

Programs can exchange information through files, inter-process communication, or networking, which you always need to investigate as potential bug sources. The program communicating with yours may have changed and started providing incompatible data, and the underlying medium may affect your bug as well, even if only temporarily. A fluctuating connection or one that corrupts your data can change or expose bugs.

Time may also have an influence on bugs. An erroneous time zone or device clock could cause problems, or a condition in the code could cause issues to occur only around midnight. One famous example is the Y2K bug, which manifested at the turn of the millennium because it was common practice before then to just store two digits designating a year.

These external interactions introduce randomness since you can't control or predict what the other end will do (see "Remove Randomness" on [page 69](#)).

Study the Environment

When you've ruled out any program-side issues, check the underlying platform or hardware. Those components could have bugs themselves or could cause them through their interaction with your programs. If your software worked before, an update or change to the environment may have introduced a bug without you even being aware of the modification.

The operating system, including its drivers and other hardware abstraction layers, sits in the middle of your platform. On top of it are abstraction layers, such as virtual machines for languages like Java and C#. Below it is the hardware, which consists of several different components: CPU, graphics card, HDD/SSD, RAM, mainboard, or maybe all of them together as an integrated system on a chip (SoC). If one of them or a link connecting them fails, it causes problems for your software.

Running out of disk space or hitting a storage quota can keep the program from working normally. The same goes for the network if your program relies on one. For instance, a weak Wi-Fi signal can drop or delay data.

Verify Your Assumptions

When questioning everything, it's essential to provide answers. As soon as you've found something that you assume may potentially relate to the bug you're investigating, you must systematically go through the areas you've newly identified as possible culprits and check for connections.

A great tool for this is, of course, your debugger. If you have the source code of the component you think may be responsible for your bug, you should start debugging that to verify your assumptions (see "Step Through the Code" on [page 100](#)). It's also useful to instrument the code (see "Instrument the Program to Report Problems" on [page 62](#)) and use tools such as a calculator to confirm intermediate results (see "Read the Code" on [page 113](#)).

As you can't always debug the potentially problematic area, you'll have to use other approaches as well. For example, if you think that your HDD or RAM may be corrupting your data, you can run dedicated tools to test them for hardware issues. When you're dealing with a product someone else provided, you might also want to run your findings by support.

Searching the internet is always worthwhile (see "Search the Internet" on [page 47](#)) when you're using externally available software. If that doesn't

help, try the documentation. Sections titled “Known Issues” or “Errata” are not unheard of and may provide some very useful hints.

Test every potential culprit and every component in isolation. Build a minimal test case for software that uses just the part giving you trouble (see “Build a Minimal Repro” on [page 210](#)). For hardware, swap out the suspicious component and test your program without it. Also, try putting the component into another device and see if that allows you to reproduce your bug in a different context.

Try running another program with a similar purpose and requirements on the same hardware to check whether that exposes the same bug. It’s possible your program just pushes the hardware over the edge because it’s so demanding or uses some features that expose a problem. Identifying commonalities between similar programs can also point you toward the bug’s cause.

Learn About the Context

A thorough understanding of your program’s context and how the individual pieces function gives you a better foundation for spotting when something isn’t quite the way it should be. You’ll see issues earlier without having to go and verify each individual assumption.

You should study all the relevant parts that underpin your program:

- How does the hardware work? What are the properties of all the device’s individual components?
- What kind of pitfalls and known issues does the operating system have?
- What other processes or programs usually run on the device that may interfere with your program?
- What networking protocols does the program use? What are their advantages and disadvantages?
- Which system was your program built with? How does that influence it?
- What programming language, framework, or library was the program built with?

Spend some time upfront to learn what makes your program tick, and you'll have an easier time figuring out why something goes wrong when it does.

17. Experiment

How? Formulate hypotheses about your program or your bug and test these theories, often by modifying the program.

Why? Testing hypotheses helps your understanding of the bug. You can observe how the bug changes if you modify its dependencies or environment and optimize your workflow by performing experiments that help the reproduction. The results may produce insights you can't easily get in any other way.

To some extent, experimenting during investigation resembles playing with the bug during reproduction (see “Reproduce the Bug Yourself” on [page 15](#)), but at the source code level instead. These experiments serve the same goal—determining what modifications have which effect on the bug. Because you need to know the bug's location or at least a related area to make exploratory changes, this strategy works best a bit later in the debugging process.

Formalize your approach and work with hypotheses that you define in a structured manner. Testing them one at a time allows you to gain more insights, each taking you a step closer to the bug's cause and a viable fix.

Conduct Exploratory Code Modifications

Experimenting with the code around your bug is easy to start. You can make almost any change you like, as long as the code still compiles. Keep an eye on what you're modifying; it might teach you something about your bug.

Ideally, you already understand the code around your bug. If you don't, read through it first (see “Read the Code” on [page 113](#)). If you can't do that, change the code to an implementation you can easily understand. This is your first experiment and enables all the others.

Experimenting helps you separate the code that causes the bug from the code the bug merely depends on. You can remove code or move it around to

learn more. Try to make the bug go away while making sure the program still works, or try to make it happen more often. If you know which lines are responsible, you can also write a unit test for the function.

When a bug always happens under certain circumstances, you should ensure those circumstances are always true. It's often useful to remove any randomness (see “Remove Randomness” on [page 69](#)) and see what happens.

Experimental changes have a positive side effect: Because you put the emphasis on the code that exhibits problems, you often end up discovering how to reach it faster and more easily. That improves your workflow and shortens iteration times.

TIP: CHECK ALL COMBINATIONS

Sometimes experimenting means just changing things in a semi-random manner. If you only have a few combinations to work with, you can quickly check all of them and get to the right one.

The order of matrix multiplications, for example, often comes up in graphics programming. Matrix multiplication is not commutative: Different orders have different outcomes and potentially even different dimensions. Getting the order wrong could make the program display distorted 3D objects, misplace them, or not show them at all.

If you only have two matrices, though, there are only two potential orders. Even three only give you six different ways to multiply them, which is still realistic if your iteration times aren't too long.

While experimenting, you can quite naturally end up implementing a workaround for the bug. This probably won't be the cleanest or most reliable solution, but it might suffice while you keep looking for a better one (see “Deploy a Quick Fix” on [page 201](#)).

Refine and Test Hypotheses

Once you've done some initial exploration, you should start structuring your process. Come up with a list of viable experiments and ask the following questions for each one:

1. What is the hypothesis?
2. How are you going to test it?

3. Once you've proven or disproven the hypothesis, what can you learn?

Having a clear structure behind your hypotheses enables you to more easily test them, repeat them, and reliably analyze what you learned (see “Keep a Debugging Log” on [page 140](#)).

Formulate Hypotheses

Sometimes, coming up with a hypothesis is hard. At other times, they simply arise when you look at the code. Certain patterns, functions, libraries, and algorithms simply ring an alarm bell when you've seen them enough.

TIP: THINK OF WELL-KNOWN PROBLEMATIC FUNCTIONS

Any C function that has the potential to write an arbitrary number of bytes makes me suspicious. I always double-check whether there's enough space in any buffer passed as a target pointer to functions like `scanf()`, `memcpy()`, `strcpy()`, `strcat()`, and so on.

Code can conceal bugs if it's too complicated to understand or adequately test. If you find the complexity overwhelming and you do know the requirements and possible inputs, you should refactor the code to be simpler. Make sure to replicate the way the existing program works and avoid adding any new bugs. You might produce a simpler implementation that doesn't exhibit the bug anymore, which lets you compare results and gradually modify your new implementations to be closer to the buggy original. Keep an eye on the code's behavior while making the modifications so you can pinpoint the one that reintroduces the bug.

You can also step back from the individual instructions and look at your code on the function level. Evaluate which ones are integral and leave out everything else. By selectively calling into specific functions, try to pinpoint the issue to one that requires further investigation.

Consider edge cases. For instance, when iterating over a container, check whether your iteration really includes the first and last elements. You should also test any operation that touches a range of memory to make sure it handles all bounds correctly.

When it comes to calculations, examine extreme values such as 0, negative values, and very small or large numbers. Such values may put extraordinary requirements on an algorithm or equation, especially when considering the limited precision computers allow numbers to have.

Check all control-flow instructions like `if, else if, else`, and so on to see whether their conditions are sensible and correct. Also, check parallel constructs—whether an `else if` accompanying an `if` does all the things it needs to do if the other case doesn't happen, for example.

You can hypothesize about your program's execution environment. A particular platform or device may influence the bug. Other programs that yours communicates with, such as remote servers or databases, may also have something to do with it. You can even test these hypotheses without changing the source code.

As soon as you realize you can affect the bug during one of your tests, you'll get even more ideas. For example, once you know that not calling into a specific function makes the bug go away, you know that this function contains the bug or at least is a prerequisite for it.

When you have a hypothesis, try to fix the bug in the hackiest way possible. If you can do it, you've found something related to the bug's cause. Then you can continue looking for a proper and clean solution with more confidence.

Confirm or Reject Your Hypotheses

Go through your list of hypotheses and evaluate each of them separately. Experiment to discover whether a particular assumption is true and, through that, learn more about your program, the code, and the bug.

Test with as little effort as possible. Take shortcuts and don't worry about following best practices when writing code if you have to do so at all. What counts now is to perform each test as quickly as possible. Of course, you must be careful not to add any new bugs, but it's also important to not spend too much time polishing temporary code.

You can test hypotheses in several ways depending on their types. Some require direct changes to the code, while you can test others by modifying values through the debugger. You can even evaluate some hypotheses without touching the code, just by using the program differently or tweaking its configuration.

One test involves changing a variable at runtime. You can do so using the debugger or hardcode the variable to some fixed value at compile time. Such a change allows you to determine whether your bug depends on the variable holding a specific value.

IN PRACTICE: VERIFY HYPOTHESES BEFORE MOVING ON TO A PROPER FIX

A project I once worked on a while ago used a custom file format for 3D meshes. At some point, the meshes wouldn't appear correctly on the screen. I eventually hypothesized that the program was reading the wrong vertex count, compared that to the one the mesh-authoring tool showed, and found that they differed by a factor of 3. It seemed the tool confused triangles with vertices.

To test this hypothesis, I changed the rendering code to multiply the vertex count by 3, which did indeed fix the issue. Then I reverted the change in the renderer and went to the mesh loading code to make sure the problem didn't occur outside the loaded file. I changed the code there to always use the vertex count I had determined for a specific mesh. Relaunching the program confirmed that this modification fixed the issue. That made me very confident, and I fixed the bug in the mesh exporter properly.

With this approach, you can also force control-flow statements to always take particular paths so the program always or never executes the code that depends on a condition. If your bug always occurs without any randomness after you do this, you've learned something and improved your workflow. You no longer have to simply hope for the right values to come up.

Similarly, you should feed hardcoded values to functions that might relate to your bug. You can simulate different inputs and observe how a function behaves when it receives them. If these values have an influence on your bug, that's a great clue.

You should also try to restructure related code. Move sections around to make relevant parts closer to each other and easier to debug. Call the code as early as you can during your program's startup so you don't need to navigate deeply.

You may even go as far as replacing the code that exhibits the bug and writing a new implementation. That means you're essentially refactoring

the code. The difference between the two versions should tell you more about your bug.

Ideally, the new implementation will be more straightforward and less optimized but use simpler algorithms, making it easier to understand, examine, and debug. You can start by rewriting a small section rather than the whole module, class, function, or algorithm, then expand if that doesn't affect your problem. If the new version doesn't have the bug, you should start by comparing behaviors to find out where the original goes off track. From there, you can try to discover the bad data's origin (see "Follow the Data" on [page 122](#)).

You can also call different functions that perform similar actions. For example, you may want to check whether a container is empty by comparing `getSize()` against 0 instead of using the `isEmpty()` function. Similarly, to iterate over all elements, you can get each element by its index rather than using iterators.

Different abstraction layers generally have distinct ways of working. If you were to send an HTTP request to a remote server, you would probably do it using an HTTP library by calling a function that wraps everything nicely so you only have to provide the URL. You could talk to the underlying socket library directly and create an HTTP request by sending a TCP packet to the correct host and port with the HTTP request as the packet data. Several layers of abstraction probably exist between these two extremes. Each layer has its potential pitfalls and trade-offs and might contain its own bugs.

Another experiment involves removing pieces of code that seem irrelevant to see if they affect the bug. This doesn't necessarily mean deletion: You can comment the code out with block comments like `/* ... */`, or conditional compilation like `#if 0 ... #endif`. If you have to actually delete any code, you should use your version control system to get the original back once you're done.

Alternatively, you can add code like `print` statements or assertions that allow you to observe what the program does (see "Instrument the Program to Report Problems" on [page 62](#) and "Debug with `printf`" on [page 158](#)), or code that checks your hypothesis itself. If you have an alternative

implementation of a function or algorithm, you could execute both and compare the results programmatically.

Learn from Example Hypotheses

Consider this example code:

```
#include <stdio.h>
float calculateAverage(int arr[], int size) {
    int sum = 0;
    if (size == 0);
    {
        return 0.0f;
    }
    for (int i = 0; i <= size; i++)
    {
        sum += arr[i];
    }
    return (float)sum / size;
}
int main() {
    int numbers[] = {1, 2, 3, 4, 5};
    int size = sizeof(numbers) / sizeof(numbers[0]);
    float average = calculateAverage(numbers, size);
    printf("Average: %.2f\n", average);
    return 0;
}
```

Now, when running the code, you might notice that the result is always 0. Let's test a couple of hypotheses:

- **Hypothesis:** `main()` doesn't correctly calculate the number of elements.
Test: Change `size` to a hardcoded 5.
Result: Nothing changes.
- **Hypothesis:** `calculateAverage()` doesn't iterate over all elements.
Test: Debug the function or add `print` statements to the loop body.
Result: The loop is never executed. The problem must sit somewhere between the beginning of `calculateAverage()` and the loop. Taking a

very close look, you might notice a stray semicolon after `if (size == 0)`, which is to blame for this bug.

If you run the program with that bug fixed, you might observe that it crashes occasionally and, most of the time, still doesn't give you the correct results. Let's test a few more hypotheses:

- **Hypothesis:** The `calculateAverage()` function is to blame.

Test: Comment out the function call.

Result: You'll see that, while you don't get the correct result, the program no longer crashes.

- **Hypothesis:** The loop in `calculateAverage()` is the problem.

Test: Comment that out.

Result: The program doesn't crash anymore.

- **Hypothesis:** The loop accesses invalid memory.

Test: Like before, debug the function or add `print` statements to the loop body.

Result: The loop body is executed one time too many. A careful look at the loop condition shows that it should be `i < size` to get the correct number of iterations.

Restore Your Code to Its Original State

When performing experiments in the code and thus changing it, you have to be very careful that you can undo your modifications. You're not trying to make exploratory code suitable for release, so its quality is very much questionable. Sure, you may have learned something from the experiments, but you'll need to implement it in a clean and reliable fashion.

If you've found a bug fix, you should keep the dirty version of it while you implement the clean one. That way you can constantly compare the two versions and ensure the cleaner one is at least as correct as the solution you found during experimentation.

You can take two distinct approaches to producing revertible changes:

- Use clearly marked comments to indicate experimental modifications. Make sure to use a string you can easily search for so you can find all changes in the whole program without going through it manually (see

“Instrument the Program to Report Problems” on [page 62](#)). It can also be helpful to add comments explaining what your changes add to the experiment in case you need to return to it later.

- Track experiments with your version control system. If you have a clean local state when you start experimenting, all the changes you make will show up in your version control system’s status overview. From there, you should be able to easily review and undo any modifications.

Using the version control system is more reliable because it won’t let you forget to mark your changes. However, if you have other local modifications already, you won’t be able to discern the changes afterward.

You can also use markers, which you can check into version control if you have to share your experiments with someone else. This becomes especially helpful if you want to share a temporary workaround that fixes the bug so you can unblock other people.

Using markers also lets you run multiple experiments in parallel without mixing them up. You can more easily revert one and keep the others around.

18. Keep a Debugging Log

How? Write down anything you’ve tried during the investigation. Add data, images, and so on to have them readily available. Also write down your other ideas and those that others suggest to better structure and prioritize your next steps. Store all this data in case the same or a similar bug appears.

Why? A log of what you’ve tried already makes it less likely you’ll repeat work or lose your place. It also lets you share exactly where things stand when you ask for help or hand the bug off.

Track Your Progress

Make sure to record everything you do by adding all the little details about your attempts to a document you update continuously. Also, write down all the places you’ve checked and tagged as unrelated. Note which hypotheses

you’ve already refuted (see “Experiment” on [page 134](#)) and what you’ve been able to rule out without dedicated experiments.

The document should be easily accessible and discoverable, so choose a name related to the bug and its description. While keeping track helps you with your investigation now, it’s entirely possible that it’ll be useful later too.

Share Findings with Confidence

You can forget what you’ve done in a bug investigation if you haven’t worked on it for a while or if you’ve been diving deep into one particular lead for several days. Sometimes it’s hard to keep track of all lines of inquiry, and you might scroll past the same code sections so many times or try the same feature in so many slightly different scenarios that all of your exploration blends together.

Having recorded your previous work protects you from stumbling into an idea multiple times and repeating the same processes in all these scenarios. You can always go back and find where you’ve left off.

Records also let you share your findings with a lot more confidence. When you’re talking to someone else and they start asking you about your experiments and results, you can give them more concrete and precise answers (see “Ask Colleagues for Help” on [page 79](#)).

Additionally, if anyone asks you for a status report, you’ve got all the information ready to go. You can give them an overview of what you already know and share in-depth data and discoveries. This could help your colleagues give you new ideas or recommend other team members to talk to.

IN PRACTICE: DOCUMENT YOUR KNOWN ISSUES

After an update, an online platform I worked on initially ran fine, then stopped responding to any requests. We had no clue what was going on until one of our users reached out and told us that opening the knowledge base triggered the issue.

That’s when we remembered we had observed this issue during development. The program would create a circular dependency between some knowledge base entries and thus send itself into an infinite loop whenever someone looked at a particular topic. The solution was to run a specific database command after each deployment.

To avoid this issue in the future, we wrote extensive documentation for ourselves, covering the symptoms in detail and making it very clear what command we needed to run. While an automated fix would have been better, we never encountered this issue again afterward.

When you document investigations and even fixes, you'll have a head start if a similar bug comes up or the same one occurs again.

Make Logging a Habit

Keeping a log of what you've tried isn't very difficult in itself. However, you must stick to your notetaking procedure and select the right information to record.

Taking notes should become a habit, which you can achieve in different ways: sticky notes on your monitor, repeating calendar alerts, or reminders about documentation in daily meetings.

Choosing the right information is easier to formalize. In general, write down anything you investigate in the process of examining and fixing your bug.

When you start looking into a new area or following up on a hypothesis, your first step should be to write down in detail what you're doing and what you hope to learn. As you gain new insights about whether your idea is relevant to the bug, add your findings to the document. Make sure to include the date and time when you record information, the project it relates to, and your name if you're keeping a shared log.

Record the conclusions for each idea you've finished looking into. If they're related to the bug, consider them very important and keep following up on them. Record unrelated ideas as well, even if all you learn is that you can rule them out.

Include Supporting Data

In addition to recording the idea in detail, as well as any facts and findings connected to it, you should add any other firsthand information you can gather. This could include screenshots, video recordings, and copies of input and output data, all of which may prove useful later when you can combine it with new insights or context. Copies of input data allow you to

reproduce any test cases you've tried more easily, while recording the output data lets you see how it changes after you modify your program.

Sometimes it's worthwhile to add segments of code to the document. In the context of other related data and observations, the actual code may provide a new perspective or important background information about the program's behavior. Make sure to enter the segment source, filename, line number, and the current version (as specified by your version control system).

Visualizations, such as flowcharts or diagrams, can often provide insights and present information in a succinct and readable fashion. You can create visualizations on paper, a physical whiteboard, or a virtual whiteboard such as Miro (<https://miro.com>).

You might also want to revisit the Unified Modeling Language (UML) and look into what kind of diagrams it offers. Different diagram types can help visualize your program's structure and behavior. For example, you can show your classes' inheritance hierarchy or how two programs communicate over the network.

Code can generate visualizations too. If your program produces data that's relevant to the bug, it might be able to visualize that data or at least make it easier to convert. Generating comma-separated values (CSV) files in code is a simple way to do that. Just write any values you want to visualize to a text file with the correct delimiters. Then import that file into a spreadsheet application and quickly generate diagrams. If your data is formatted correctly, you can even copy and paste it directly into Excel without the intermediate step of writing to a file.

Summarize How You Fixed the Bug

Once you've fixed the bug you're working on, make sure to note the solution. Explain how you reached it and any other important details or observations. For instance, if you already know that the implemented change has potential weaknesses, you should add them as comments in your code but also write them down in your documentation.

Describing the bug fix will be very useful if you should ever run into the same bug or a similar one. Any additional information will be essential if you need to apply the fix again or to improve it.

Add your documentation to the bug notes in your team's project management tool. That way, it has an immediate connection to the actual report and anyone else can see the information if they're dealing with the bug. You can either insert your notes directly into the report as comments or add a link to your document.

Avoid Distractions

New ideas come up every now and then as part of your investigation. These can be anything from a new angle on a bug's potential cause to a new area you think is worth exploring, and they come from a variety of sources. Co-workers might suggest them when you're discussing the bug or mention something remotely related in other conversations, the program's documentation might inspire you, or they might just hit you when you're busy with other tasks. No matter what the inspiration, don't follow up on the idea immediately. Switching contexts between multiple investigations increases the time you need to refocus and often means you forget essential details. Follow one lead to the end and determine whether you can learn anything from it before you dive into the next one.

Track and Prioritize New Leads

Instead, write down each and every idea as well as your thoughts and potential hypotheses. It's okay not to structure them at first as long as you record enough details that you can find out what you meant later. A keyword may have many different interpretations, so you need to be specific.

When you start tackling a new lead, it's time to clean up your notes. Start with properly spelling out what the idea is about so other team members can potentially understand it. As with hypotheses, it's useful to attach your ideas to the bug in your project management tool.

Prioritize each idea based on the effort it takes to investigate and the probability that it will lead to valuable new insights about the bug. Tackle the most promising one first because, when debugging, it's usually crucial to get results quickly. Leads that take weeks to follow make great contingencies, but it's better to prioritize those that require only minutes or hours.

A list of relevant leads helps you remain confident even when the one you're working on doesn't pan out. It also reduces the time between evaluating ideas because you don't need to brainstorm or search once you're done with one.

TIP: ALWAYS HAVE A PLAN

Having a list of ideas to try and leads to follow always gives me a certain peace of mind. It's like always having a backup plan. I especially make sure to have at least one good idea lined up on Fridays so I don't have to worry the whole weekend about what I'll do come Monday morning.

Having a prioritized list of leads means that you can easily share the work with your colleagues. In the event you need help or someone else needs some work to do, you can simply give them your list or parts of it. Even if your notes don't provide all the information, at least you'll be able to fill your co-workers in if they ask.

Use Documentation Tools

After figuring out what you should write and where you should store it, you must decide how to do so. Generally speaking, a more straightforward solution is better because it comes with fewer dependencies and it's easier to get started, but sometimes you need the features an advanced tool offers.

Plaintext files let you read and edit your notes with nearly every tool on the platform and are easy to diff text files. However, you can't add data in other formats to them. Markdown-formatted text files are similar, but they offer standardized formatting that lets you highlight certain aspects of your notes. You can also link internal resources and data in markdown-formatted text, although you're still responsible for managing it. The format offers no validations or guarantees that external data is valid or even existent.

Binary formats like *.docx* (Microsoft Word), *.odt* (LibreOffice), and *.one* (Microsoft OneNote) let you directly embed other files, including videos and images, and other types of data. They integrate everything, and anyone can view them without having to open external files.

Online platforms on the internet or inside your organization can store your documentation too. Obviously, everyone on your team will have an

easier time accessing internet platforms: Internally hosted documents might require VPN connections, or the links in them might move around or contain temporary host names, making them unreliable. Like binary formats, online platforms provide extensive formatting options and let you embed images and videos. Online hosting also offers easy linking and helps create connections between the different tools your organization uses. You can also easily share it with others and configure permissions to make access simpler.

A great example of a convenient online platform is a wiki. Wikis can be shared easily and edited collaboratively, and they offer history tracking. Popular offerings include Confluence and Notion, but you can also host open source alternatives like MediaWiki.

Ultimately, which tool you use comes down to a trade-off. Text is easier to handle, binary is more powerful, and online solutions are simpler to share. Try to follow the standard your team or company uses. If you don't have one, consider taking the lead and establishing it. It'll make everyone's lives easier if they don't have to switch back and forth between different tools and formats.

Keep an Archive

Once you've gone through the effort of taking notes while fixing a bug, you shouldn't just throw them away. Instead, store them for later reference, since the same bugs or similar issues often pop up. Also, sharing this kind of knowledge can help others learn from it.

Try to store your notes in a central location, following any guidelines your team or company may have. You may want to access your documentation from several different machines where you work, get access from a meeting room, or return to your notes after upgrading to a new computer. Central storage also allows others to see your notes and put the knowledge to good use.

A history of all the changes to the documentation shows how it evolved, which may make it easier to follow the investigation's progress. This is a lot simpler when working in a text format. Storing the notes in one of your version control system's repositories can help you and others remember to record their history. Wikis and online platforms usually have

history recording already built in and highlight differences between the individual versions when they display the records.

You or your colleagues must be able to find relevant information in the archive of previous investigations. That's why you should reference your notes from the bug report they're related to: That's usually the first place anyone looks. Additionally, you should make sure the team knows that your documentation exists and where to find it. Include the bug ID, its description, and relevant keywords in the notes to help people find them with full-text searches.

Use Chats as a Log

With the proliferation of remote work, teams increasingly rely on asynchronous communication in the form of chats and instant messaging clients like Discord, Microsoft Teams, and Slack. These services store most of your communication in user-accessible history and even make it searchable in most cases.

Valuable information can end up in chat history for several reasons. Some people are very chatty about the project they're working on, some like to share what they've learned, and some reach out when they're stuck. If you have people like this on your team or if you're such a person, you're lucky: They're already creating documentation. While all these notes aren't very well structured or even attached to the bug, they still exist. Using the chat service's search feature, you should be able to uncover relevant information if you need it.

You can also start documenting your progress on your chat platform. You should always do this when you've asked a question about your bug and received the solution in some way other than a reply in that chat. Otherwise, someone might find a description of your bug without a solution, a classic situation that's always annoying.

19. Build Your Own Tools

How? Build your own debugging tools, tailored to your program. These tools can be anything from a code snippet that checks a piece of data to a full graphical interface for inspecting your program as it runs.

Once you've built one, share it with your teammates so they can use it too.

Why? A tool you build yourself can show you exactly what's happening inside your program, such as logic and data that existing tools can't reach because they don't know your code. A custom tool fits your program's specific systems, workflows, and quirks, and it can catch problems automatically before they ever become bugs.

Implementing custom tools has the same purpose as using existing ones (see "Use Off-the-Shelf Tools" on [page 164](#)), but you can tailor them to your program, particularly in relation to your use case, workflows, and users. You can also incorporate specific rules or budgets.

[Table 3-1](#) provides an overview of the types of tools you can implement.

Table 3-1: Types of Implementation Tools

Tool type	When to use	Pros	Cons
Integrated	For quick solutions or tools most users need	◦ Comes with the program and can use its features for input and output	◦ Can be difficult to control ◦ Limited by the host program and device
External	For bigger tools or those specialists sometimes need	◦ Can be changed and executed independently ◦ Can run on different (more powerful) devices ◦ Can use different languages or technology	◦ Requires inter-process communication ◦ Missed easily
Hybrid	When you need features of external tools but don't have the capacity to develop one from scratch	◦ Can use some features of more powerful devices	◦ Can't be changed or distributed independently

The following sections go into detail about how you can implement these tools in your program.

Implementation

The effort required to implement a tool and its level of sophistication can vary drastically. A tool can be a code snippet that verifies the integrity of a piece of data, a full-blown GUI tool that accepts mouse and keyboard inputs with many tweakable options, or anything in between.

Tools can be very generic and make your debugging life easier, but they can also be very specific to a certain feature or even exist to prevent or solve only one particular bug. Balance the effort of implementation with the bug's severity and the tool's general usefulness. Invest more time in building custom tools that find critical bugs than for one-offs or cosmetic issues.

Follow two guiding principles:

- It should be as easy as possible to build new tools. Whenever you want to add one, you shouldn't have to worry about the window layout or how your program can trigger it.
- Your tools should be easy to access. A hidden tool no one knows about doesn't help anyone debug anything.

That's why you should have templates and defaults for your debugging tools. If they all basically behave the same, developers and users will know what to expect.

Integrated Tools

You can integrate tools into your program in several different ways. First, you can build in processing for textual commands, which is similar to a command line interface. While this is a very versatile and powerful way to offer generic functionality, it's also not very user-friendly, so it's more suited to programmers.

A better solution is to dedicate a GUI to debugging and development. You can implement it with the toolkit you're using to build your program or with one that separates the debugging process. Keep the GUI accessible from the main program so any developer or tester can easily reach it.

If your program renders its graphical output itself, using an immediate mode GUI (ImGui) toolkit is a great choice. Dear ImGui (<https://github.com/ocornut/imgui>) is widely used and easy to integrate. It lets you add a powerful debugging GUI on top of your existing program, enabling quick discovery and allowing user input to handle a wide variety of tasks.

You can also implement hidden input commands that are triggered by secret key combinations or a click on a specific area, but compared to GUI solutions or the command line interface, hidden commands are less user-friendly.

NOTE

While command lines and hidden commands won't ever be easy to discover, you can work to make them more accessible. For example, you can provide documentation, teach your colleagues, or implement a more intuitive user interface. While these steps involve extra effort, they let you be more selective about who you expose to which tool.

Integrated tools have the advantage of being easily accessible, because you don't have to refer anyone to another place or program. However, the host software and device constraints often limit their capabilities, meaning they can't always visualize everything you'd like to see or accept all the inputs you'd like to provide.

External Tools

External tools are nimbler than integrated ones. Because their codebases are smaller, you can change them more easily and iterate on them more quickly. They also present less risk to your program, since they're not attached to the part you're shipping. You can always change external tools even when you've also frozen the program's code for the next release.

Working with an external tool means running another program in parallel with the one you're debugging, so they need to interact with each other somehow. They can establish inter-process communication, share memory, work through pipes, or use existing message queues. The programs can also communicate via shared files, which doesn't lend itself to real-time exchanges but works well for providing inputs like initial setups.

Another communication method involves setting up a network connection, which is flexible because you can use it both on the same device and with a remote machine. In the latter case, you can connect a tool on an accessible or powerful computer with a program on a device with limited input or output capabilities, such as a smartphone.

Additionally, an external tool is a lot more independent. You can run it when you need it or just let it be. If you set it up right, you can even start it after the main program.

You can write an external tool in another programming language or use different middleware from your main program, which may make sense when requirements and goals differ between debugging tools. One might prioritize ease of development over performance, for instance.

Be sure to tell people when you're building and distributing a tool, and inform future users that it exists.

IN PRACTICE: RUN EXTERNAL TOOLS AUTOMATICALLY

On one project, we had to iterate on many assets constantly for most of the development time. We began by defining a lot of guidance and rules for naming them as well as their technical requirements and limitations, then decided to implement all of the rules and requirements in an external tool that would validate what we created.

This allowed us to create assets in several different tools without running them through our main program and to automatically validate all files anyone added to our version control system. This significantly reduced the number of bugs from assets that didn't conform to our standards.

Hybrid Tools

Many tools mix the two approaches, sitting partly inside your program and partly outside it. For instance, you can integrate a web-based interface into your program for debugging and development. Such a setup has many of the same advantages as an external tool: Because the frontend is outside your program, it can also visualize any input that works with a browser. Access to the debugging features still requires a separate program, though, which makes them hard to find.

Using existing libraries, you can easily incorporate an HTTP or HTTPS server into your program, generate HTML that displays what you want to

show, and accept inputs via HTTP. If you need more advanced web technologies, feature-rich libraries also exist for your programming language.

Recommended Features for Debugging Tools

The more complex the program, the more features the tools to debug it should have. (You may want to implement tools in complicated systems early, maybe even during development.) The particular tools you should provide naturally depend on your program, but these examples should give you some ideas of what to include.

Command Line and Backend Programs

You often encounter a few standard tools for command line programs and other software that doesn't have a visual component. One is *verbose mode*, which forces the program to print a lot of information, helping you retrace its actions. Output from verbose mode could give you a hint about a bug's root cause or potentially even tell you about it outright.

Another tool is the option to perform a *dry run*, in which the program operates normally but skips any actions that have a permanent effect. For example, a program might not write the results of its calculations to a database during a dry run.

Some command line programs run as servers, and a server usually detaches and keeps running in the background once it has started. That's convenient in normal use, but it makes debugging harder, because you can't see what the program is doing. In that case, you can usually tell it to stay in the foreground of the terminal where you launched it, so you can watch its output and interact with it directly.

Embedded devices face the same challenge from the other direction. Internet routers, for example, are little boxes running on embedded hardware that manage traffic between the internet and your local network, and that's about it. Yet almost all of them offer a web-based interface where you can diagnose issues and adjust settings.

Offering debugging tools is a bit harder for libraries integrated into other programs. They have no control over what they receive or produce, but they can provide external tools or interfaces that let the main program forward input as text or graphics. You can implement such interfaces by

deriving from a class, or with callbacks or delegates. The main program registers its own functions, and the library calls them whenever it has something to share.

IN PRACTICE: PROVIDE TOOLS FOR YOUR LIBRARIES

When developing an open source library to handle input processing in C++, I ended up building two tools to ease development and debugging. One was an interface that the main program could use to display debugging information directly on the screen. It could show what keys users pressed and where the program was registering mouse clicks or finger taps. This method thus gave us immediate feedback and told us whether all inputs were working correctly.

I also built an external tool that could connect to a running instance of the library via a network. It could display active inputs and their values, record all actions and play them back later, and create graphs visualizing how continuous values changed over time. Many other features like this made interaction with the library easier.

GUI Programs

Programs with a GUI can also work for debugging. It's often easier for them to provide visual feedback and accept inputs because they generally have the concept of a certain window or input field being in focus. For example, if you were to open a second window dedicated to debugging, it could remain open alongside your program's main window as long as the software runs. A common feature in networked programs, especially in video players and call clients, is the ability to display live network statistics like the current transfer rates, total amount of data transferred, and connection quality. Such information makes it easier to tell whether a given problem stems from a network issue or a bug in the program.

Another useful debugging tool is launch mode specification. For example, a command line argument could specify that a program should run in a single-threaded mode without any concurrency at all, or an argument could make it use a specific toolkit or graphics API instead of the default.

Game Development Features

Debugging tools have a long history in game development and often end up in the hands of users. Cheat codes, for instance, don't need a visual component: Entering secret strings in the right contexts activates the

debugging mode, which can change users' game experiences. Another common tool is a built-in console where you can enter text commands and view output data. Finally, an on-screen GUI overlay enables accessible and easy-to-implement debugging.

You should provide certain baseline tools to your testers and fellow developers. Allow them to check out any game content quickly, whether that means levels or equipment. Also, being able to play the game without dying (the good old "god mode") is always useful when testing.

Visualizations showing the game's performance are also very common. You should be able to display how many frames per second (FPS) your game achieves at any given time and what that means in terms of milliseconds. Ideally, the times break down into different areas like logic, rendering, and so on, because any of these can become a bottleneck and it's useful to see which is taking a while. An on-screen performance display helps you identify issues early and quickly and gives less technical people an idea of how the features they use will affect performance.

Debugging tools may also visualize data differently. They could show the collision geometry of the level you're playing or the draw calls, which the game sends to the GPU as a list, so you can more easily tell if something is rendered.

TIP: IMPLEMENT THESE COMMANDS IF YOU'RE DEVELOPING A GAME

The following commands are common in games, and they've proven useful in both debugging and development:

- Load a level
- Start a mission at a checkpoint
- Play a cutscene
- Fake progress through the game up to a specific point
- Move the player to a particular position
- Provide a certain piece of equipment
- Spawn a specific nonplayer character (neutral and enemy)
- Activate invincibility mode
- Kill all enemies
- Finish the current mission successfully or unsuccessfully
- Advance to the next checkpoint in the current mission

Of course, you probably can imagine many more features that might make your life easier, but this list is a good starting point. For example, if you need to own a specific piece of equipment to trigger a bug, having a command to make that happen will save you a lot of time.

Existing Tools

You can also integrate existing tools. Some programs offer APIs or standardized file formats to help you incorporate powerful debugging features. For example, RenderDoc and VTune offer APIs that you can use to talk to these tools and to provide more context about whatever you want to debug.

Promotion and Documentation

Once you've implemented a new tool, ensure that the target group can use it. Let them give it a go and ask for feedback. They'll have a bunch of feature requests and ideas for modifications that would make the tool even more effective for them.

Make your new tool easy to access and discover if you can. Sometimes debugging tools need to be a bit hidden so they don't distract from the program itself, which means developers also may forget about them.

Use multiple communication channels, such as email, chat, and meetings, to spread the word about what your tool does and how anyone can find it. Repeat the announcement from time to time so people have your tool in the back of their minds if they get into a situation that requires it. Document your tool thoroughly. Anyone on the team should be able to use it without needing you to personally give them an introduction. Once you write the documentation, adjust it every time you tweak the tool.

Release Considerations

When you get to the point of releasing your program to the public, you'll have to decide which debugging tools to keep and which to strip out. For anything you don't want users to have, disabling it in public builds while keeping it for internal use is usually enough.

Whether a tool is safe to ship depends on what it can do. A debugging tool might hurt performance, expose private data, or even give some players

an unfair advantage in a networked game. Because of risks like these, most teams keep their tools out of public releases. But it's worth weighing case by case: A well-chosen debugging tool can also help users diagnose and fix problems on their own.

20. Build a Minimal Repro

How? Strip the bug down to the smallest program that still triggers it, either by building a fresh minimal version or by cutting everything unrelated out of the full one. The goal is to have something small enough to test quickly across devices and easy to hand to someone else.

Why? A minimal example clears away the noise of a large program, so what's left is almost entirely relevant. Because it's small, it will build and run fast, tightening every loop of your investigation.

You'll often want to create a minimal repro if you're having trouble finding a bug in a large and complex program. Different areas may seem to cause the bug, depending on where you're looking, or you may have trouble debugging or understanding all the components.

If you need to ask external support for help, you'll most certainly need to have a minimal repro. Complexity, size, and legal concerns may keep you from providing your entire project to the support team, or they may refuse to deal with it because understanding or reproducing the problem would take too much effort. If a third-party library is among your list of possible culprits, you can also use the minimal repro to convince the external developer that your bug is real, and their software is a likely cause.

Choose a Minimization Method

Minimal repros must include all the elements that you need to reproduce the bug and nothing else. In the best case, the bug appears as soon as you start the program.

You'll also need a recovery strategy. Ideally, you can set up a copy of your program and use it to remove everything you don't need. However, some projects might need to be located in a specific spot, or are too large to fit on the disk twice, or take too long to duplicate. In those cases, your

version control system might let you revert any changes as soon as you're done.

TIP: AUTOMATICALLY GENERATE MINIMAL REPOS

You can automate aspects of creating minimal repos. For instance, Nvidia's Nsight Graphics debugger (see "Use Off-the-Shelf Tools" on [page 164](#)) gives you the option to generate a C++ capture when debugging a graphics frame. This creates a completely new C++ project, exactly reproducing the frame you were looking at.

The new project is entirely self-contained, without any dependencies on your program or assets. To reproduce the frame, the code sets up any tasks and performs all the graphics API calls in the correct order. It even converts assets like textures, shaders, and meshes to code and includes them with the project.

Such a C++ capture removes all the complexity from your CPU-side code and any data or third-party libraries you're using. If you can still reproduce your graphics bug in the capture, you can be sure it exists somewhere in your interaction with the graphics API or the assets for this particular frame.

When you contact Nvidia with a graphics problem, they usually ask for a capture like this. It makes their lives a lot easier because they know they'll be able to understand the code and reproduce the bug.

Generally, you can create your example either by starting from scratch and copying any relevant bits from your main program, or by taking your main program and incrementally removing things.

Start from Scratch

To start from scratch, create a new project in the same programming language and framework as your main program. It should also use the same tools for building and debugging and target the same platform.

You could also base your work on a template or sample, which makes the most sense when evaluating whether a bug is related to a third-party component. Take a working sample with that component to ensure everything is set up correctly. Using a sample gives you confidence that you're working based on a known good starting point and lets you avoid implementing some of the boilerplate functionality.

Make sure that your empty project compiles and that you can start it. Then begin introducing features and assets to the program in a step-by-step

fashion, adding only what you strictly need to reach the point where you can reproduce the bug.

First, add the libraries, SDKs, and frameworks that your program is based on and that are involved with the portion you need to reproduce the bug. If you don't need graphics for the repro, don't integrate anything related to them. If you don't need a network connection, skip it. Once you're done, make sure your program works as expected up to this point.

Then start bringing over the source code from your main program. Keep adding incrementally and repeatedly check whether everything works as expected or you can already reproduce the bug. If assets, configurations, or other external files are involved, you must also bring those over. Whenever the source code requires some data, copy it from the main program and check whether it behaves as expected.

As soon as the bug occurs, you fulfill the reproducible requirement, but you still need to verify that the repro is indeed minimal. If all the features you've added build on top of each other (you can't remove one without breaking its dependencies), you've successfully minimized the repro. Cut anything you still can, but always make sure you can reproduce the bug.

If you have a program that deals with a lot of assets or data coming in from the outside, you should also try a hybrid approach. Use a minimal version of the code with the regular assets and data, or minimize data and assets in the main program. Being able to reproduce the bug with one of these setups tells you very clearly that it must be related to either the code, the input data, or the assets.

Starting from scratch is a good approach if you don't need to set up too much boilerplate or infrastructure to reproduce the bug outside your main program. Using an existing template or sample has the advantage of meeting the third-party developer on common ground. They'll have an easier time understanding the reproduction and following the changes you made to run into the bug.

IN PRACTICE: TEST FOREIGN FONTS IN A DEDICATED PROGRAM

A project one of my teams was working on supported many different languages, which meant it also had to support Chinese and Japanese characters. While the team had set everything up and implemented support, the QA team reported that the text became unreadable at some point.

The developer started investigating the issue inside the main program. He quickly realized that iterating on it would require him to build the whole project, deploy it to the target device, and test it there for every little change.

Instead, he created a new test program. He integrated the middleware for text rendering into it and added code to produce Chinese and Japanese characters. As a result, it didn't take him long to track down the root cause and configure a proper chain of fallback fonts.

Starting a new project to look at a particular feature or system in isolation is also very useful for developing new functionality. It's only possible if that functionality isn't going to be tightly integrated with existing systems, but if you're creating something relatively independent, starting a new project lets you iterate more rapidly. Once you have a working version, you can start integrating it into the main program.

Start from Your Current State

You can also start with the existing state of your program. Confirm that you can indeed reproduce the bug, then get rid of any code you're sure won't affect the bug and continue with increasingly smaller steps, regularly testing whether the bug is still there. When you eliminate code that depends on an asset, go ahead and remove that as well. Make sure it won't be loaded, and if it might be picked up automatically, delete it altogether. The same goes for external libraries or middleware.

Removing something from the project doesn't necessarily mean you have to delete it outright. You can simply disable code and assets, which makes it easier to bring everything back later. However, be careful to make sure you've disabled an element properly.

When disabling source code, you just have to comment out the code that shouldn't run anymore. You can easily remove large chunks by commenting out a function call or system instantiation. Move assets and other files the program might pick up to another directory so it doesn't find them anymore.

IN PRACTICE: SIMPLIFY THE OUTPUT

In one augmented reality (AR) project, I had to render a 3D object on top of images coming from a camera. This sounds like a simple enough problem, but it involved

many dependencies: camera interaction, 2D images to fill the screen, and a facility for rendering 3D objects, which I integrated using a third-party framework.

The goal was to have a rendered 3D object align precisely with a particular real-life object the camera captured. Markers let my software identify the target easily, but the 3D projection wouldn't match up with the real-world equivalent. It was always a little displaced or improperly rotated.

After investigating the issue for a while, I decided I needed to quickly iterate changes to the projection matrix to determine whether I could even influence the projection and how I should adjust it to match the captured images. I started removing everything unrelated to this core problem: all the logic giving the user a good experience and the 3D object rendering. I only left the line rendering in 3D space to visualize the projection. Ultimately, this allowed me to fix the bug because I could just play around with the values in the projection matrix, derive some insights from that, and figure out what the underlying math did.

As soon as you get to the point where you can't reproduce the bug anymore, you've identified a cause. Whether it's the root cause remains to be seen, but at least now you have a clue about what can affect the bug. Closely examine the portion you had to remove. Take it apart even further to bring parts of it back and leave others out. You should be able to isolate the bug's root cause this way.

Minimize the Input

So far, this strategy has been about dealing with programs that you can modify, but it also works when the code isn't yours or you don't have the source. In such cases, the minimal reproducible example is the input data you provide.

Assume one of your files makes a certain program crash or misbehave. Create a minimal reproducible example using the previous steps. You'll need some intuition and guesswork to figure out the relevant parts, but if you approach this task systematically, you should be able to determine exactly what input data provokes the bug.

You can send a minimal reproducible example in file form to the developer of the program you're having trouble with. They'll be very grateful to you for making their lives easier, because their bug should be pretty straightforward to fix now.

Share the Example

If you haven't found the bug's cause yourself with the minimal reproducible example, you've now got a great tool you can use when asking your colleagues or third-party support for help (see "Ask Colleagues for Help" on [page 79](#)). The example makes communication about the bug much easier and often more fruitful. Whoever needs to reproduce the bug doesn't have to learn anything about your program's architecture or how it's used. They can just start investigating with a focus on identifying the bug's cause. If you suspect the bug's root cause lies in a third-party program, having a minimal reproducible example also helps convince someone that the bug is actually in their software instead of yours.

Test It on Another Device

Before sharing your example, try it on another device because that's exactly what the other person will do as their first step. If the example doesn't work on their device for whatever reason, they won't be able to help you and likely won't bother to investigate it further. By testing your example on another device, you can also rule out whether it's related to your normal development hardware. If the bug occurs on other devices, you're good to go. Otherwise, you now know that something about your machine triggers it. That's when you should start investigating yourself. Still, you can certainly check with whoever you intended to ask for support whether they know anything about issues related to devices like yours.

Share Source Code, Not Binaries

When reaching out for external help, provide only the sources of your minimal reproducible example in a form the person helping you can analyze and modify. Never share binaries for analysis or debugging. They most likely won't work, they take a lot of storage space, and they aren't useful at all to anyone who needs to take a deep dive into the example.

WARNING

Remove any proprietary or private data from the sources you're going to send to a third party so you don't leak any confidential information. Such information might be about platforms and their details that not everyone can access, or it could include personal data, passwords, or authorization keys.

While the minimal reproducible example should be as complete as possible, you really can't include things like the hardware and operating system. You also might not be able to share hardware peripherals or other programs that run simultaneously. Describe anything relevant that you can't send to the person helping you.

Prevent Regressions

In addition, it's helpful to have a minimal reproducible example to prevent the bug from coming back into your program. Use it as a unit test once you've fixed the bug and make sure it can be easily executed as part of your test suite.

Of course, doing so requires a relatively clean implementation. Just deleting code or moving files around won't do the trick in this case. Writing a new minimal program from scratch is the best idea.

21. Debug with printf

How? Modify your program to print text to standard output. The text should contain relevant data about what your program is doing. Consider integrating a dedicated logging library if your program produces a lot of text.

Why? Printing text output is one of the most versatile debugging methods. You don't need a debugger, and it works in most situations.

Adding `printf` statements to your code helps you understand what your program does. Such statements can also be called log statements. Using them is a form of instrumenting the code (see “Instrument the Program to Report Problems” on [page 62](#)) that allows in-depth analysis. While a regular debugger (see “Step Through the Code” on [page 100](#)) is usually the best choice for the task, it's not an option in some situations.

The name `printf` comes from the C function, which is the traditional way to print text to the standard output channel, and the *f* means “formatted,” because you can print more than static text. This function makes it easy to insert additional information.

NOTE

Even though this strategy's name comes from the C function, virtually all programming languages and platforms have similar features. C++ has the idiomatic `std::cout << "Text"`, C# uses `Console.WriteLine()`, and Python offers `print()`.

Debugging with `printf` shouldn't be your default method, as it does have downsides. If you know what those are, however, it's a great backup solution. Sometimes finding and setting up a debugger is more hassle than simply adding a couple of print statements. The choice all depends on how much time you spend on the project and how complex you expect the bug you're working on to be.

Avoid printf Pitfalls

In contrast to other methods of debugging that don't require any modifications, you have to change your program's code for this one, which produces a couple of side effects.

First, modifying the code means you'll have to build it afterward. This may be a negligible point if you're using an interpreted language or debugging a small program, but it can have a considerable impact on larger codebases. Once you've built the code, you must deploy the program unless you're debugging it on your local machine (see "Shorten the Feedback Loop" on [page 19](#)).

Second, changing the code can have an effect on the bug itself. It's possible, and not as uncommon as it may sound, that adding print statements makes the bug appear more rarely or disappear altogether. That's actually an excellent clue because it means there's something fishy going on (see "Remove Randomness" on [page 69](#)), but it keeps you from learning more with this method.

Whenever you want to follow up on a new branch in the program's logic or realize you need to verify the data in a particular variable, you must modify the code and add new print statements. This is a very inflexible way to debug, and changing direction or looking deeper into a lead comes at a significant cost. Ideally, you'll have the foresight to add print statements to all potential leads so you don't have to go back and insert them incrementally.

Writing print statements takes a certain amount of effort. You need to know how to print the data you want to examine, and that's not always straightforward with complicated structures. Then you have to add the print statements to several spots in the code to understand the logic flow and how the data evolves over time.

Another potential downside of `printf` debugging is that it can generate a lot of text in a little time. If you print a value once per frame in an interactive application, you may end up with 60 outputs per second or more. Large amounts of text can cause problems with the observation interface and make it difficult for you to find any relevant information. Avoid printing more than one or two lines or calling your print function too often.

Know When to Use printf

Use `printf` when a regular debugger won't help you, which can happen if you're working in an environment that lacks an available debugger because of its programming language, framework, or target device. If that's the case, you don't have any other options.

If the debugger isn't reliable or takes a lot of effort to get up and running, you'll need to decide whether to set it up or fix the bug in a more cumbersome way. Keep in mind that you may have to fix bugs in the future without a debugger as well, so using print statements in those cases may make more sense.

IN PRACTICE: USE PRINTF FOR DEBUGGING IF IT'S EASIER

I once worked on a project where most of the logic was written in C++, but the whole GUI was implemented in Lua. While debugging the C++ side was a breeze, bugs in the GUI posed a much greater challenge. Lua had some debuggers, but with the bespoke C++ integration, we assumed setting them up would take quite a while, so we spent a large chunk of the project debugging the GUI using only `printf` statements. At some point, one of our developers took a deeper look at one of the debuggers and sat down to integrate it, which didn't even take that long and did help us with a couple of bugs. We ultimately still did most debugging with `printf` statements, though, because the external Lua debugger was always one more step away.

Using a debugger might also change the program's behavior and influence how the bug is reproduced. While you can easily identify and

prevent some cases, such as when the software adjusts its logic if a debugger is present, in other cases, attaching a debugger may affect execution timing and be harder to deal with (see “Remove Randomness” on [page 69](#)).

Sometimes using print statements provides an advantage even when you can use a debugger. If you need to examine a complex data structure that your debugger can’t visualize, for instance, writing some logic to print it is easier than clicking through nested structures. Doing so gives you more control over data visualization and lets you stick with the language you usually work in.

Another benefit might be when you want to observe how your data evolves over time. Printing a variable’s value at different points shows you how it changes. You can draw conclusions from just looking at the values, or you can paste them into another program to visualize and analyze them.

Implement printf

While you probably already have some print or log statements in your program, you can improve them for your debugging process. Once you’ve identified code related to the bug, you should inspect it closely (see “Read the Code” on [page 113](#)) and look for spots where you can’t easily derive the logic flow or a variable’s contents. That’s precisely where you need to add print statements.

Because the cost of building, deploying, and starting a program is usually huge compared to that of adding a single print statement, you should add those print statements liberally. You can always scroll past them when analyzing the output.

Even if you add print statements all over the place, you won’t be able to find your bug’s root cause on the first try. You’ll typically need to run the program, examine the output, and spot holes where you’re missing information. Then add new print statements to address those holes and start again from the top. Keep doing this iteratively to close in on the bug.

Print Concrete Data

Make your print statements very specific to their spots in the code. Print any concrete data that might be relevant to the bug. The closer a print

statement is to a likely bug location, the more detailed its message should be. You can roughly categorize print statements into three tiers:

- Print something you can search for later. This could be any string, but use short, unique messages that don't show up anywhere else in the code, so you don't get many false positives when you search for them (for example, `xxx` or `!!!`).
- Print a description of what the code does. To do this, you need to read the code and summarize its intention so you can write a brief statement about it. Some examples might be `Creating database...` or `Unable to connect to the server; will instead use the local cache.`
- Print actual data and useful information, like concrete variable values. Because you need to determine how to print the data, these messages need to be more complex. For example, `20 out of 480 entries in program.ini were invalid.` or `Server example.com responded in 21s: Unable to find the file.`

Different programming languages have different ways to format messages. In C, for example, you must specify the exact type of the variable you want to add to the message.

Printing the contents of linear lists, arrays, or vectors is straightforward. Just iterate over the elements and execute one print statement for each. Printing a map is more advanced because you have to print the key and the value, and they can be of different types. Also, the order of a map is not necessarily defined, so you might want to sort it first to make the output more readable.

Nested containers are even more complex. It's not unheard of to have a list inside a map inside another map. Because those containers are properties of classes, you may not be able to see these indirections, but they're very real. When it comes to such complex compositions, you may want to consider providing a dedicated print function, which goes through the hierarchy and transforms it into a string representation in a format like JSON.

Add Metadata

Include some metadata with your print message, such as:

- The source filename and line number where it originates: With this information, you can quickly jump to the right place in the code. Many languages even support adding this data automatically.
- The severity of the message: Common categories are (in increasing order of severity): verbose, information, warning, error, and fatal.
- The category of the message: Define broad categories that relate to the systems making up your program. This allows individual developers to focus on messages that are relevant to the area they're working on.

You should also add an explicit identifier to your messages so you can group them. For example, add the bug number like `Bug42: Trying to close the file`. Doing so helps you find all print statements related to your examination, which will come in handy when you need to revisit some of those spots or when you've fixed the bug and want to clean up after yourself.

Leverage Advanced Logging

A variation of `printf` debugging is logging, which usually relies on more advanced features and is even enabled in release builds. Logging systems can usually deal with many more messages than standard output setups and automatically include useful metadata by default.

If you only rarely use `printf`, your program and the language's standard print mechanism should provide acceptable logs. Be aware, however, that printing a string to standard output or error channels is all that mechanism does. You have to implement message filtering and saving yourself and define how to add categories, severity, or filenames to messages.

Evaluate Logging Libraries

You can also integrate dedicated logging libraries, which have many advanced features for long-running, complex programs. Servers, for instance, run for a long time and deal with a lot of requests, and users may have to debug any issues without access to more advanced tools.

Logging libraries offer many features out of the box to help you generate and handle complex messages. These include metadata addition, formatting options, and message routing and filtering. If you find yourself

relying on `printf` debugging often, consider dedicated libraries and custom solutions. These are a few of the more commonly used, freely available libraries:

C++

Boost.Log (<https://github.com/boostorg/log>)

spdlog (<https://github.com/gabime/spdlog>)

Quill (<https://github.com/odygrd/quill>)

C#

log4net (<https://logging.apache.org/log4net/>)

NLog (<https://nlog-project.org>)

Serilog (<https://serilog.net>)

Python

Loguru (<https://github.com/Delgan/loguru>)

structlog (<https://www.structlog.org>)

The standard logging module (<https://docs.python.org/3/library/logging.html>)

Some services also provide log storage and feature-rich viewing. If you're working on a project that incorporates a large chunk of server logic, consider integrating such a service to help spot patterns and problems.

Choose a library that fits your program's goals and complexity. For example, smaller command line programs are fine with printing to standard output, while servers need complex libraries.

Use Logging in Production

In contrast to `printf` debugging, production environments and binary builds usually have logging enabled when they reach users. To keep logging from affecting the user experience, they don't generally record the more detailed messages, but warnings and errors can often give you an idea about what's going wrong. While verbose messages deliver more details that make it easier to determine a bug's root cause, they may offer more data than you can manage.

You have two options for gathering logs from end users: Ask the users to provide data or automatically record and share it. When asking users, store all production build logs in a file so they can provide theirs and get

ideas about the issues they deal with. Libraries that handle crash dumps often have automatic log storage (see “Collect Crash Dumps” on [page 91](#)).

Make sure to protect user privacy when you record log output by making messages filter out sensitive information like passwords or keeping them from recording it in the first place. Depending on the applicable laws, even data like the user’s IP address may be subject to regulations.

Clean Up

Once you’re done with the bug you’re working on, it’s time to clean up after yourself. Review all the changes you’ve made to the code and decide whether to leave them in or revert them before committing the code to your version control system.

It’s important that you don’t generate too much noise during normal operation so you and your team will notice exceptional behavior. Also, if you change any of the code around your `printf`, you’ll probably have to update the print statement itself, making your code a little more inflexible. You must also consider your program’s performance. No print statement is free. Having several print statements together will slow down your program at some point, even if it’s highly optimized.

In general, leave in any print statement that provides useful information in an error case, as well as anything you can mark as a warning, an error, or an even more severe alert. Useful statements contain concrete data, while simple progress output about control-flow decisions or function calls is rarely relevant. Leave in more verbose statements or messages only if you can keep them from affecting runtime unless severe problems occur. Don’t enable such messages by default.

Some programs allow runtime filtering via configuration or command line options (see “Build Your Own Tools” on [page 146](#)). You still need to be careful with the runtime cost, but it can be helpful to offer verbose messages on demand so your users can debug at least some of their issues by themselves.

TIP: USE ZERO-COST LOGS

In many compiled languages, you can express print statements so that they don’t generate any code at all if a problem isn’t severe, which can keep them from dragging

down production builds. Just enable verbose messages in debug or development builds, or on demand when you need to address a particular issue.

Here's how you can do it using the C or C++ preprocessor:

```
#define LOG(severity, ...) \  
    if (severity >= GlobalLogSeverity) \  
        printf(__VA_ARGS__)
```

If severity is lower than `GlobalLogSeverity` and you compile your program with optimizations, you won't generate a single assembly instruction for this code.

You probably clean up print statements after successfully fixing your bug, which makes a lot of sense. However, that means you're changing the code again. Be careful when removing log output and test your bug fix afterward.

Store a version of your code in a safe place before you start undoing your print statements so you can retrace your steps later. If your changes reintroduce the bug, you'll have an easier time figuring out how that happened.

22. Use Off-the-Shelf Tools

How? Use existing tools to examine aspects of your program related to the bug.

Why? Such tools may alert you to any potential bug sources in your program. They also let you inspect opaque systems and data so you can spot otherwise hidden problems.

Your debugger is the most versatile tool at your disposal (see “Step Through the Code” on [page 100](#)). The tools in this section are more specialized, so each will do a better job with bugs of a given class. I don't cover all of the available tools here, but they should give you some ideas of where to look for what you need.

Consider Availability

Generally speaking, tools fall into three categories:

Open source or free Tools available without any barriers at all. Any developer on the project can try them.

Paid Tools that are low cost but require getting approval, which can still hinder adoption.

Enterprise solutions Tools that are so expensive their utility must explicitly justify the purchase.

Some tools may be available only to certain groups of licensed developers. Getting a license may be time-consuming, require coordination with others, or costly even if the tools themselves are free.

Know When to Reach for a Specialized Tool

The tools mentioned in this section aren't necessarily designed for debugging, but they all can help you explore specific areas. They may also draw your attention to other bugs (which may not be as critical), so be careful to keep your attention on the one at hand.

A general tool hands you raw data and leaves you to make sense of it, which can make it hard to draw conclusions. A specialized tool works from the same data but does more with it: structuring it, highlighting what matters, or interpreting it for you. It's worth reaching for one when an investigation hits a wall and you can't see the information you need.

General-Purpose Tools

Many tools enable you to investigate data or other aspects of a program more deeply than you could otherwise. While debugging may not be their most common use case, the following tools can certainly help with it:

Diff and merge tools

When you need to compare files, some tools can highlight differences you wouldn't see with your bare eyes, like hidden Unicode characters, stray whitespaces, or discrepancies in line endings. Some diff tools even support comparing directory hierarchies. Examples include Beyond Compare (<https://www.scootersoftware.com>), diff (common command line utility), KDiff3 (<https://github.com/KDE/kdiff3>), P4Merge (<https://www.perforce.com/products/helix-core-apps/merge-diff-tool-p4merge>), and WinMerge (<https://winmerge.org>). While most

tools compare text, Beyond Compare and others can also handle images, and hex editor HxD works on binary files.

Disassemblers

These tools can help you analyze your program's compiled code. They show instruction execution and potential issues. If you need to examine compiled software without a debugger, you should look into a disassembler. Examples include Ghidra (<https://github.com/NationalSecurityAgency/ghidra>), IDA Pro (<https://hex-rays.com/ida-pro>), and `objdump` (a common command line utility).

File search tools

By indexing all your disks, these tools find any files by their names in an instant. If you need to locate specific files during your investigation, such tools will be very valuable. Examples include built-in file search (for example, Spotlight and Explorer), as well as Everything (<https://www.voidtools.com>).

Format validators and linters

You may have to make sure a given blob of data conforms to the format you expect. Some formats have fixed specifications for tools to check, while others are so generic, they represent a multitude of formats, in which case schemas are useful and describe valid data. Examples include HTML/CSS validators, JSON Schema (<https://json-schema.org>), and XML's document type definition (DTD).

Fuzzers

Fuzzing means testing the program's input space with unexpected or malformed inputs that may trigger undesirable behaviors. These tools aim to uncover security vulnerabilities, such as buffer overflows, format string vulnerabilities, and injection attacks. Examples include AFL++ (<https://github.com/AFLplusplus/AFLplusplus>), Honggfuzz (<https://github.com/google/honggfuzz>), and libFuzzer (<https://llvm.org/docs/LibFuzzer.html>).

Hex editors

If you need to examine binary data, a hex editor will be your tool of choice. Editors usually present the data in hexadecimal but also include readable text. Many text editors have a hex editor mode integrated, but

dedicated tools also exist. Examples include 010 Editor (<https://www.sweetscape.com/010editor/>) and HxD (<https://mh-nexus.de/hxd/>).

Screen recorders

Recording what happens in your program can be useful for creating bug reports, pinpointing differences in behavior between multiple executions, or examining what happens during reproduction. Examples include OBS Studio (<https://obsproject.com>) and ScreenToGif (<https://nicke.tech/screentogif>).

Screenshots

You can also capture still images of your screen to show a bug's symptoms and compare different program outputs. Examples include the built-in shortcuts (SHIFT-COMMAND-3 or -4 on macOS, PRTSCN on Linux, PRTSCN or WIN-SHIFT-S on Windows) and Greenshot (<https://getgreenshot.org>).

Text editors

A text editor is most likely already part of your standard set of tools. Consider whether you want to get a more powerful editor or one with more features. Examples include EMacs (<https://www.gnu.org/software/emacs/>), Notepad++ (<https://notepad-plus-plus.org>), Sublime Text (<https://www.sublimetext.com>), vim (<https://www.vim.org>), and Visual Studio Code (<https://code.visualstudio.com>).

Code Playgrounds

Playgrounds are online editors where you can enter your code and share it with others. They usually also support compilation and let you inspect the result, and some offer code execution with observable output data.

Playgrounds offer you a way to experiment with the code and iterate on your ideas or prototypes in a very short amount of time. You can try out changes with minimal effort and easily share what you've created, so you can concisely communicate your problem and share ideas or solutions. Many different playgrounds exist, each focusing on a different language or technology:

Compiler Explorer/Godbolt (<https://godbolt.org>)

This tool allows you to enter code in many different programming languages, compile it, and view the generated assembly instructions. It offers a wide variety of different compilers and versions.

Rust Playground (<https://play.rust-lang.org>)

You can compile and execute Rust code here.

Shadertoy (<https://www.shadertoy.com>)

This tool lets you create and share shaders, which you can execute in your browser to show off the results.

SQL databases

You can use these databases to create a schema in the browser and then run queries against it. Examples include SQL Fiddle (<https://sqlfiddle.com>) and DB Fiddle (<https://www.db-fiddle.com>).

Web development

These tools primarily focus on web development, allowing you to write HTML, CSS, and JavaScript code. You can run what you create right in the browser to inspect the results. JSFiddle (<https://jsfiddle.net>) and CodePen (<https://codepen.io>) are among the examples.

Static Analyzers

Static analyzers check your source code or the compiled binaries without executing the program. By examining the code's structure, syntax, and data flow, they can identify potential programming mistakes, vulnerabilities, and other issues that could lead to bugs. This helps you spot problems without specifically looking for them. One type of static analysis is your compiler's warnings. The compiler will alert you to potential problems by default, ideally pointing you toward solutions. When investigating a bug's cause, you may want to enable all possible warnings for the project or at least the relevant areas. For GCC and Clang, you can enable any relevant warnings using the command line parameters `-Wall -Wextra -pedantic`. You may want to skip `-pedantic` because it doesn't always point you toward useful issues.

When working in MSVC, you have to use `/Wall` instead. However, this might be too verbose, so in many cases `/W4` works better.

If you have a relatively warning-free build, you should enable warnings as errors. You can do this for GCC and Clang using `-Werror` and for MSVC using `/WX`. Doing so means the code builder will fail whenever it encounters a warning.

Compiler-independent static analyzers include:

Cppcheck (<https://cppcheck.sourceforge.io>)

Popular for C and C++ programs, it focuses on identifying undefined behavior and code that might cause bugs.

Black Duck (<https://www.blackduck.com/static-analysis-tools-sast/coverity.html>)

A powerful but expensive tool that covers most languages.

PVS-Studio (<https://pvs-studio.com/en/>)

This commercial offering supports checking C, C++, C#, and Java. It supports a wide range of platforms and can identify many bugs.

Pylint (<https://github.com/pylint-dev/pylint>)

This tool for Python checks for errors, code style violations, and potential bugs. It enforces coding standards, identifies unused variables, and performs various code quality checks.

Roslyn Analyzers (<https://learn.microsoft.com/dotnet/fundamentals/code-analysis/overview>)

Tools that provide static analyzers for C# programs.

While it's helpful to run static analysis locally on your development machine, the ideal setup operates several static analyzers regularly, because different ones may uncover different problems with your code. You should incorporate analyzers into your continuous integration setup and inform people immediately if a check starts failing after they change the code.

Code Sanitizers

Sanitizers, which are becoming more commonly available in various compilers, instrument the compiled code to add runtime checks and collect information during execution. They can detect common programming errors that may lead to crashes, vulnerabilities, or incorrect behavior.

Many different areas have specialized sanitizers:

AddressSanitizer (<https://clang.llvm.org/docs/AddressSanitizer.html>)

This program detects memory errors such as use-after-free, buffer overflows, and out-of-bounds accesses, identifies memory corruption bugs, and provides detailed reports on problematic access. Supported by Clang, GCC, and MSVC.

Control Flow Integrity (<https://clang.llvm.org/docs/ControlFlowIntegrity.html>)

By ensuring that your program follows valid paths, this tool protects against control flow hijacking attacks. Supported by Clang.

DataFlowSanitizer (<https://clang.llvm.org/docs/DataFlowSanitizer.html>)

This sanitizer helps identify tainted data and track its flow, so it's useful for detecting security vulnerabilities such as command or SQL injections. Supported by Clang.

LeakSanitizer (<https://clang.llvm.org/docs/LeakSanitizer.html>)

You can use this tool to detect memory leaks and discover information about the leaked memory blocks. Supported by Clang.

MemorySanitizer (<https://clang.llvm.org/docs/MemorySanitizer.html>)

This program helps identify uninitialized memory use and provides reports on its sources. Supported by Clang.

ThreadSanitizer (<https://clang.llvm.org/docs/ThreadSanitizer.html>)

Because it detects data races and synchronization issues in multi-threaded programs, this sanitizer helps identify concurrent access to shared data and reports race conditions. Supported by Clang.

UndefinedBehaviorSanitizer (<https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>)

This tool detects undefined behavior in your code. Supported by Clang and GCC.

Networking

A lot can go wrong when two programs need to communicate via a network. Many different networking protocols work on various layers of interaction, and specialized tools address most of them:

Browser developer tools

Most browsers, including Chrome, Edge, and Firefox, have integrated tools that allow you to inspect HTML, CSS, and JavaScript. They also support debugging JavaScript code, analyzing network requests, and tracking performance. You should use these when building a web application.

Fiddler (<https://www.telerik.com/fiddler>)

A widely used web debugging tool, Fiddler allows you to capture, inspect, and modify HTTP/HTTPS traffic. It acts as a proxy server, sitting between the client and the server, and provides a wealth of features to analyze and troubleshoot web requests and responses. Try this tool if you have trouble with your HTTP or HTTPS requests.

Postman (<https://www.postman.com>)

Postman is a tool for interacting with web APIs that you can use to manually make requests, verify your understanding of the API, and ensure that servers respond correctly when you're debugging them. If something goes wrong with your web API, Postman can help you verify the results manually.

Wireshark (<https://www.wireshark.org>)

With Wireshark, which supports many platforms and protocols, you can inspect network packets. If you can't find any other tools to debug a network protocol, you should use Wireshark and check for TCP or UDP packets.

Graphics

Graphics processing units (GPUs) are harder to debug than other software, so dedicated tools let you inspect what happens on the GPU and how your program interacts with the graphics API. They're usually specialized for either a single hardware manufacturer or a particular graphics API. Here are a few examples:

Graphics APIs

Tools that target graphics APIs support debugging the commands and data you send to the GPU. They let you analyze frames, examine shaders, and view intermediate rendering results. Examples include

PIX on Windows (DirectX12, <https://devblogs.microsoft.com/pix/>) and RenderDoc (Vulkan, <https://renderdoc.org>).

Hardware manufacturers

With tools specialized for hardware manufacturers, you can inspect what happens on particular companies' graphics cards. They include frame debuggers that display the rendering process in fine-grained detail and tools for inspecting memory use and shaders. Examples include AMD Radeon GPU Profiler (<https://gpuopen.com/rgp/>) and NvidiaNsight Graphics (<https://developer.nvidia.com/nsight-graphics>).

Validation layers

Newer graphics APIs such as DirectX12, Metal, and Vulkan support instrumenting the code to catch usage errors and potential bugs. They don't enable instrumentation layers by default, because those degrade performance, but you can enable them when you run your program.

IN PRACTICE: INSPECT RAW DATA USING TOOLS

During the development of a new rendering backend, I noticed that parts of the rendered output differed a little from our specifications. I used the frame debugger for the target platform to confirm that all the draw calls were there and executed in the correct order.

When I compared the output to the result from a platform where the backend was already working, I noticed the lighting was slightly off. Checking the normal maps for each platform verified that they were different. To make sure that the graphics API or texture loading weren't responsible, I debugged the loading code on the CPU, adding a breakpoint right after the texture data got read into memory. By copying and pasting the data into a temporary file, I was able to open it in PVRTexTool (<https://developer.imaginationtech.com/solutions/pvrtextool/>).

Comparing the image to the original input confirmed my suspicion that something was wrong with converting normal map textures for the new platform.

Some more debugging proved the root cause was a data processing step for normal maps that the new platform didn't execute correctly. If I hadn't been able to view raw texture data with a graphics debugger, I might not have been able to identify the bug.

Memory

Problematic interactions with memory are a widespread source of bugs in all languages, particularly C and C++. That's why specialized tools like

those below address potential memory issues:

dmalloc (<https://dmalloc.com>)

This library for C and C++ helps detect memory allocation and deallocation errors. If you have difficulties identifying memory leaks or writing out-of-bounds, it can be useful.

memory_profiler (https://github.com/pythonprofilers/memory_profiler)

This tool enables you to view a Python program's memory use and identify potential leaks.

Valgrind (<https://www.valgrind.org>)

This commonly used tool features memory and thread error detection with a focus on C and C++, and the developers say it'll work with programs in any language.

VTune (<https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html>)

This powerful Intel tool has a lot of different features, including assistance tracking down memory leaks.

System Tools

Some operating systems are minimal and simple, while others can do a lot but produce complicated interactions. Tools like the following help you with bugs where the cause lies somewhere between the system and your program:

Dependencies

Some bugs are related to missing DLLs on Windows, and these tools can help you determine which libraries a given executable depends on. Examples include Dependency Walker (<https://www.dependencywalker.com>) and Dependencies (<https://github.com/lucasg/Dependencies>).

Disk space

Sometimes you have to find out what files are eating up your disk space or why your project has a specific size. These tools allow you to identify large files and directories easily. Examples include altWinDirStat (<https://github.com/ariccio/altWinDirStat>), WinDirStat (<https://windirstat.net>), and WizTree (<https://diskanalyzer.com>).

File locks

When a file is locked, the operating system prevents it from being deleted or overwritten. A tool for investigating file locks can help you determine why. Examples include File Locksmith (<https://learn.microsoft.com/windows/powertools/>), Process Explorer (“Find Handle or DLL”), and Windows’s built-in Resource Monitor (“Associated Handles”).

Logs

Most logs end up in the standard output but not all. Operating systems have system-wide logging capabilities and, in some cases, you need to check these for messages relevant to your program. Examples include DebugView (<https://learn.microsoft.com/sysinternals/downloads/debugview>), `dmesg` (Linux), and the `/var/log/` folder (Linux).

Operating system tracing

Some tools allow you to monitor what your operating system is doing and how your program interacts with it, which is necessary for tasks like reading or writing files, dealing with processes or threads, and network communication. Examples include DTrace (macOS), Process Monitor (<https://learn.microsoft.com/sysinternals/downloads/procmon>), and `strace` (Linux).

Processes

These tools give you detailed information about the processes currently running on your machine, like the full path of a running executable and which command line arguments were passed in when it launched. Examples include Activity Monitor (macOS), Process Explorer (<https://learn.microsoft.com/sysinternals/downloads/process-explorer>), System Informer (<https://systeminformer.io>), `ps` (a command line tool), System Monitor (Linux), and Task Manager (Windows).

You should check which tools every library, SDK, framework, or API you’re using includes and which ones others have developed that can help you. Such tools are often essential when working with third-party products because they may provide the only way to debug very complex or specialized programs or “black box” software that doesn’t let you see its internal processes.

Large software packages tend to contain a multitude of tools, such as those for peeking into the internals and measuring performance. Always try the most specialized ones first. If a framework-based program is running out of memory, for example, you should use a dedicated tool it provides instead of looking at the program's memory use in the process monitor.

23. Talk It Through

How? Explain the bug in as much detail as possible to a person, group, or even an inanimate object. Choose an appropriate setting to discuss the issue, and be prepared with all the information you've learned.

Why? Talking about the bug and presenting your progress in finding and fixing it makes you aware of the gaps in your knowledge and details that don't make sense. If you're talking to a person, they may also have helpful feedback or ideas.

Another strategy to learn more about your program and how it works is to take what you know already and talk about it. It's easy to get lost in the details when investigating a bug, which means you can lose perspective and spend too much time on leads that are unlikely to provide results or on trying to come up with new ideas.

Explaining a bug forces you to think everything through. (This is a bug-specific version of the Feynman technique, where you keep studying a concept until you can teach it to someone else.) It helps you identify gaps in what you know, information that contradicts itself, and places where things don't make sense. Talking it through helps you look at the bug, including its symptoms and the program's systems, from a certain distance.

This practice is different from asking for help (see "Ask Colleagues for Help" on [page 79](#)), which is about getting pointers, ideas, and solutions from others. Of course, both strategies can lead to in-depth discussions, but the initial step and the objective are very different.

Some people just naturally talk about their work all the time and gather great input that way, while others either don't feel like sharing their struggles or don't even put themselves in situations where they can do so.

Especially for introverts, it's common to forget the benefits of simple conversation about a project.

Finding the best time to reach out about a bug is a balancing act: too early and you might not have enough information, too late and you might waste your time when talking could have made an enormous difference. As a guideline, it makes sense to reach out whenever you feel you're stuck or not making fast enough progress.

Define Your Goal

When you start talking about your bug, you want to think up new ideas. (While you can hope for valuable questions and input if you're talking to a person, it's generally up to you to find new leads.) Discussing what you know can be useful in two different ways.

First, you may be able to identify information you're missing. It could be something small like how data gets from one system into another or why your program needs a certain feature.

Second, you might spot correlations you didn't see before. These can be connections between a symptom and a user input, your previous observations and the bug's cause, or pretty much anything else. Such correlations provide new insights into interdependencies with your bug and thus places to start investigating.

Once you've identified new leads, the next step is to go back into your program's sources and fill in knowledge gaps to get a more complete picture of the bug.

Choose Who You Talk To

Generally, you can talk to anyone willing to listen. Mostly, it doesn't matter if you're speaking to someone who's already familiar with the problem, someone completely unrelated, or even a random object. If you discuss the bug with someone outside the team or company, make sure not to share any privileged information or break nondisclosure agreements.

IN PRACTICE: TALK TO A COLLEAGUE TO FIND THE ISSUE YOURSELF

I can't even count how many times I've stumbled upon a realization when describing an issue to a colleague. While I'm telling them what I've observed so far, the

correlation between what I just said and the bug's cause suddenly becomes clear.

When dealing with a bug in a program that played videos, I asked a colleague to take a look. I started telling them that I had made sure the playback code looked good, the video files did exist, and the correct codec had compressed them. Then I said I'd checked videos in other parts of the program and all of them worked without problems except for the one that the bug report was about.

That's when it hit me.

I went back to my computer and checked that one video, and, of course, it had an odd resolution. While the problem really was the code not handling odd video resolutions, realizing that the one video was special was key to figuring out the root cause.

Talk to an Inanimate Object

Because you can learn different things when making yourself describe something out loud, it's even helpful to talk to an inanimate object. At least such an object is always willing to listen, and it even has the time to do so. The inanimate object in question is traditionally a rubber duck, which is why this is also known as *rubberducking*. However, any other object works just as well.

Talk to a Person

This strategy can be even more productive if you have another person to talk to. A person can ask you probing questions, provide sensible feedback, and offer suggestions about where to look next or what other areas might be related to the bug.

When talking to people with experience in the program and its sources, you can skip some basic information like the software's purpose, its general architecture, or the feature that the bug affects. Similarly, someone who's familiar with debugging as well as understanding and solving complex issues may know about common pitfalls or techniques when dealing with bugs like yours.

It's always quicker to align with someone with a shared mindset, but that also means you share common assumptions and prejudices about the program. You may both overlook details, leaving you with gaps in your knowledge, and might both have difficulty seeing the bigger picture or coming up with new ideas to investigate.

If the person you're talking to doesn't know anything about the program and its sources, they'll be more open-minded and question assumptions. Such a person can be someone from a different project, someone from outside your company, or someone higher up in its hierarchy, as long as they're not involved in the day-to-day business of interacting with and building the program.

Talk to a Group

Group settings might make you feel more pressure to get bug presentation right, which means you may have an easier time identifying gaps yourself. You also get more diverse points of view and opinions with a group.

Moreover, one person's ideas can stimulate other people to have ideas. If you're lucky, a fruitful discussion may evolve. Some of the toughest bugs stretch across different parts or systems, and getting the right people into a (virtual) room simultaneously is a great way to find a solution that takes everyone's requirements into account.

The ideal size of this group depends on a few factors. If you already know the bug touches on multiple areas, you should try to get an expert on each. Keeping the size down, however, can make some people more comfortable speaking up.

Ideally, you know the preferences and quirks of the people you'd like to talk to. One person may always take the lead, while another may derail discussions with vaguely related subjects. You should plan the group discussion accordingly, which could mean you form more than one group or leave out an expert because they might keep everyone else from exploring all potential avenues.

Plan the Logistics

When discussing a bug with someone or something, you should make sure you have time to explain it in detail with no distractions. Book a meeting in your calendar and set your status to "do not disturb."

Prepare the Environment

Once you've set aside space, you should get the environment right. It should stimulate ideas, provide the information and materials you need, and not distract from the discussion.

These requirements can easily be at odds. For example, meeting rooms are very quiet but might not give you access to the program's sources or your notes. Walking around or going outside have the same problem, though both boost critical thinking and idea generation. Sitting in front of your screen together lets you display all the information but makes you vulnerable to interruptions. Pick the option that seems to offer the most benefits to you. You can also switch between different environments to get different benefits.

Set Up for Remote Debugging

Of course, not everyone is always in the same physical location, so you may very well have to have your bug discussion in a video call. That limits your choice of environments but has some benefits. For instance, viewers can investigate the bug independently and then contribute what they've learned. You can also easily share your screen to show elements like erroneous behavior or the source code. Sharing control with other people lets them point at items on the screen and demonstrate their point with sketches or tentative code modifications.

Prepare Your Presentation

Covering everything you need may take some preparation time, depending on your ability to improvise a presentation. Assembling some notes upfront will help you remember any essential information you've already discovered. If you're keeping a log, your preparation is pretty much already done (see "Keep a Debugging Log" on [page 140](#)).

Presentation aids can also help. Visualization, such as diagrams or flowcharts, can do wonders to get a concept or point across. It's also great to have a whiteboard ready in case something comes up that's difficult to express with just words.

After going into detail about what you've learned already and what you've tried, you'll most likely get some questions. They'll either cover facts you've already checked and forgot to cover in the beginning or address some new angles of the problem. Pay attention, since a question might open up a lead, and ask questions in return.

You can use a physical or virtual notepad to record points from the discussion, but if you're talking about source code, you can also add

comments right in and know exactly which code the idea you're writing down relates to. Make sure to mark your comments so you can easily find them once you're ready to start the investigation (see "Experiment" on [page 134](#)).

Another method to verify you've understood something correctly is to rephrase what the other person has said. If you're mistaken about something, it quickly becomes obvious when you use the wrong words or misrepresent some facts.

Decide What to Cover

When talking about your bug, cover as much as possible of what you've already learned. Talk about the facts you've found and what areas you've investigated didn't yield any new information.

Start with describing the bug's symptoms and how it can be reproduced. Your description is ideally similar to a detailed bug report (see "What Makes a Great Bug Report?" on [page 11](#)). Reveal any external dependencies or interdependencies between the bug and other features or systems. Go into detail about the bug's location and what happens when you try to debug it (see "Step Through the Code" on [page 100](#)). Describe what causes you think are likely and what causes you've ruled out.

You should allow questions at any time so your counterpart can really get to the bottom of what you're saying. If you defer questions to the end, people will likely forget at least some of them.

Once you're done describing the bug and handling all questions, everyone can start the discussion. The people you're talking to hopefully have some great ideas, but you can also ask questions about areas they know. Bouncing good ideas off each other often makes them great.

TIP: BE A GREAT LISTENER

If someone is seeking your input on a bug investigation, you need to understand the symptoms, the reproduction, and what your colleague has already ruled out. Listen actively to what the other person is saying and try to refrain from interrupting or responding prematurely. If you have questions or comments, it can be helpful to take notes and then interject when an opportunity arises.

Once the initial presentation ends, ask lots of questions to complete your understanding. For anything unclear or where you see missing information, ensure that

the other person has explained everything they know. If some detail is missing, either they forgot to share it or you've just identified a new lead.

Summary

This part of the book discussed a number of strategies for examining your source code in detail, modifying the program, and gaining assistance in the form of outside resources to learn more about your bug's cause.

If you haven't been able to get to your bug's root cause with these strategies, consider going back to [Part II](#) and probing the system for more entry points.

Part IV

FIX



So far, you've reproduced your bug, probed your program, located at least one entry point in the sources, and examined it closely. Your role shifts from here, from finding the problem to building the solution, and your goal is to eliminate the bug and all of its symptoms in a way that's robust, maintainable, and stable.

The strategies in this part will help you fix your bug at its root cause, or, if that fails, through alternative means. You'll also discover how to learn from each bug so you and your team avoid similar ones later, how to apply a single fix to the same problem wherever else it appears, and why it pays to fix bugs sooner rather than later.

If you have difficulty removing the bug, you may need to go back to the previous workflow steps and learn more about it. Stopgap solutions can help if you don't have time or the ability to really understand the root cause, but understanding the criteria that contribute to a bug fix's quality will help you find the best remedy possible. The following factors can make sustainable resolution hard:

- Time pressure: If you have to release your program immediately or the bug blocks multiple people, you might not have enough time to identify

and fix the root cause.

- **Limited control:** Fixing the bug sustainably might call for changes you can't make, either because you don't have control over the relevant systems or because they'd break another part of your program. You might only be able to update one side of a network, for instance, or the bug's root cause may come from data you can't fix.
- **Need for extensive changes:** You might need to make major changes to the program in order to really address the bug's root cause. Such changes generally take a lot of advance preparation and time or may be so large relative to the bug itself that they just don't make sense.
- **Lack of expert knowledge:** The bug might be in an area that you and your teammates don't know enough about. Experts in some technologies are very rare and often busy. With older code, you may not be able to contact the experts at all, and they may not have left comprehensive notes.
- **Technical constraints:** A fix that demands more memory, processing power, or data transfer speeds might require different hardware. If you can't change the device you're working on or your connection, bug fixes that require these boosts won't work.

Any combination of the listed challenges may mean you can't implement the perfect solution. In such cases, it's okay to just address the symptoms and plan to fix the bug properly later.

Temporary workarounds range from masking the symptoms to rewriting the offending code to turning the bug into a feature. Sometimes the best move is to leave the bug alone or cut the affected feature entirely. Such approaches can keep your users happy while you work on a more long-term solution.

What Makes a Great Bug Fix

A great fix should be:

Minimal The solution changes the program's sources as little as possible.

Complete It removes the root cause from your program's sources, eliminating all of the bug's symptoms.

Permanent Your remedy keeps the bug from returning and prevents similar mistakes.

Clear The rest of your team can easily understand why and how your implementation solves the bug's root cause. In situations where that's not the case, you should document the fix and link the documentation in all relevant places.

Tested You and your QA team have thoroughly tested the solution to make sure it has the other attributes in this list.

Timely Users won't have to wait very long for the issue to be fixed.

While you'll ideally find a solution that fulfills all these requirements, that's often impossible. Timeliness conflicts with the other criteria particularly often: If the bug drives off users before you can fix it, it doesn't matter how good the solution eventually is. You should still ultimately try to discover a lasting solution, so you don't find yourself working on a related bug soon after.

Always fix only one bug at a time. The solution you commit to your version control system should contain only what's necessary to address your specific problem. Otherwise, you or others may struggle to understand the solution or to revert it if it has side effects.

Take your program's trajectory into account and make your fix future-proof. Consider whether it will work with shifting requirements such as different types or amounts of data, varying user groups, or new features.

The remedy should make your program less buggy and better behaved in general, not become a liability down the road. This goal requires you to understand the problem and implement the solution carefully so the bug can't creep back in.

Make It Reversible

Make sure you can undo your change after you've made it. Your version control system will help, but a clean, minimal change that meets the requirements listed previously will be easier to roll back.

You'll need to revert any solutions that don't work, then go back to the probe and examine steps of the debugging workflow. If the solution's side effects cause more trouble than the bug's symptoms, you should also revert it.

There's one more reason to keep your change reversible: Undoing it temporarily is how you confirm it actually made a meaningful difference (see "Verify" on [page 182](#)).

Address the Root Cause

People strongly associate debugging and programming, so you might reasonably expect to find the bug in the code. However, bugs can also come from broken assets, misconfiguration, or bad team processes and workflows. Be sure your solution addresses the real root cause of the problem you're handling.

Getting to the bottom of the issue may take some time and more iterations of the debugging workflow, but it's better to make the effort now than to risk the bug reappearing when you've forgotten parts of your progress.

Ensure your remedy corrects only the bug you're working on and doesn't affect anything else. If it has any other implications, you need to get approval from the stakeholders in the affected areas and loop in everyone who might be surprised when the program suddenly acts differently. Informing these people also lets them offer new ideas.

Verify

Once you think you've solved the bug, confirm it. Run the reproduction at least a couple of times. Then undo your change, check that the bug comes back, and reapply the change to make sure it's really what resolved the problem.

Test the change in isolation, so it's the only thing you commit to version control. Revert all your other local modifications, leaving just the one that addresses the bug, and test again on its own. That way you know the solution doesn't depend on some incidental change you made along the way.

If the bug turns out to have more than one cause, untangle the problems one at a time. A code change and an asset can both contribute to the same

bug (say, the code breaks the asset), and you'll need to correct both to keep it from coming back. Commit the two changes separately so it's clear what you modified and why.

Finally, have someone else confirm the fix once you've checked it in. Ideally, this will be the person who reported the bug, testing in the same environment. Otherwise, you risk having solved a related problem, or something that was never broken in the first place. If the tester can't reproduce the fix, the report comes back to you.

24. Just Fix It

How? Surgically modify your program to remove the bug's root cause. For a bigger change, loop in experts in other areas of the program.

Why? Cleanly removing the root cause makes your program better. Any continued development will have a solid foundation, and the bug won't come back unexpectedly.

The most straightforward way to fix your bug is to address the root cause you've identified in the previous workflow steps. It's essential to handle only that issue and avoid affecting anything else.

You need to deeply understand the bug and its origins in order to achieve a minimal and complete solution. *Minimal* means that you make only necessary changes, while *complete* requires you to get rid of all the bug's symptoms.

Remove the Root Cause

Completely removing the bug's root cause prevents the error from reoccurring and similar issues from arising and may even fix other problems. This requires you to comprehensively understand both the cause itself and how it connects to the bug's symptoms. While learning that much may be easier in smaller programs, larger and more complex ones pose more difficulties, which is why this book exists. By getting close enough to the root cause, using the strategies it discusses, you should be able to fix the one mistake that produces your bug.

IN PRACTICE: KEEP AN EYE OUT FOR SIMPLE FIXES

A library I was working on was supposed to write valid files of a specific format. Unfortunately, other software that handled these files didn't always recognize them properly. After some investigation, another team member found that the library was writing `mime_type` instead of `contentType`, which the file format specification mandated. We could implement this solution quickly, even though finding it took a while.

If that doesn't work, turn to other strategies (see “Deploy a Quick Fix” on [page 201](#) or “Find a Creative Solution” on [page 212](#)). A stopgap solution can buy time if a minimal, surgical approach requires major or fundamental changes, such as refactoring a lot of source code or re-creating complex assets. Once you've implemented that, though, go back and fix the bug properly. The right solution will make your program better and lay a good foundation for future changes. Ideally, try to fix any bugs as soon as possible (see “Fix Early” on [page 196](#)).

Change the Original Sources

You have to implement a bug fix in your program's sources, since automated or even manual steps might overwrite your artifacts and completely remove any changes there. If that happens too long after you resolve the bug, you may have forgotten about the details and have to repeat part of the debugging process. To avoid this situation, see “Keep a Debugging Log” on [page 140](#).

Making experimental changes and fixes (see “Experiment” on [page 134](#)) in intermediate artifacts may be a faster way to evaluate them. However, once you find a solution that works, you need to go back to the original sources and make the change there.

Even when you fully understand the root cause, implementing the right solution isn't always straightforward. You may need to research how to solve the problem or talk it through with colleagues, especially if it demands experience or skills you don't have, in an unfamiliar technology, process, or target platform. When that happens, loop in an expert to get to a good solution faster (see “Ask Colleagues for Help” on [page 79](#)).

Before making the change, be absolutely sure it's the right one. Mistakes could mean you don't address the root cause or even make the bug worse, and then you'll need to undo all your work.

Identify What to Change

The bug's root cause will determine how you make your changes. You may need to apply many different modifications, from changing a tiny bit of the source code to rewriting the whole system. If the program needs a feature to work correctly, you may have to add it.

However, adding features in response to a bug report shouldn't become the norm. Some people file missing features as bugs because that's a way to quickly schedule and complete tasks. Others mistakenly do so because they don't know how to request features or that they want them to begin with. Tell anyone who tries this to go through the proper channels instead.

IN PRACTICE: WATCH OUT FOR FEATURES DISGUISED AS BUGS

I once developed a tool to automate the development environment setup for a larger project. A colleague told me it didn't seem to be making any progress when installing certain software. I tracked the bug to another version of that software already being installed, which turned the bug report into a feature request: We needed the startup tool to detect and check existing software and either let the user know about it or uninstall it automatically.

First, I unblocked my colleague by talking through uninstalling the tool manually and then installing the correct version. Then I scheduled an implementation for the new feature.

Changing your program's sources could also mean correcting a noncompliant asset or one with poor technical implementation. You may need to establish new guidelines if a compliant asset doesn't work as intended.

The solution may even relate to the whole program's design or architecture. If the original design documents don't produce the right feature, or mean it doesn't work with others, you may need to change them. This could result in a lot of follow-up work making multiple changes to the code or assets.

A specification issue for a communication protocol or data format may require changes to not only the program you're working on but others that use the same specification. You may need to update stored data to match the guidelines or even write a conversion program to do so. The later you

discover such a mistake, the more extensive the changes you'll have to implement.

IN PRACTICE: BE CONFIDENT IN YOUR BUG FIX

On a project that simulated a large area with considerable distances, my team ran into floating-point number inaccuracies because we used `floats` to store coordinates and some objects would move far away from the origin. To improve accuracy, we decided to completely change the scale, basically dividing all sizes and distances by 100. This meant we had to both change code and rescale all our 3D assets.

A couple of months later, we discovered that the scale change made everything tiny compared to the default settings of most programs and frameworks we used, so we had to revert it and modify all the code and assets again. The real fix involved using a feature to reposition all our objects around an artificial origin.

If you realize your team's workflows made it easy to commit the original mistake, you should update them. Doing so may involve having a few meetings to work out better processes, checking your procedures regarding outside vendors, and revising entries in project management tools.

Collaborate with Others

When you have to implement significant changes, consider getting more people on board. As a first step, tell whoever's assisting you regarding the bug, its root cause, how you're planning to fix it, and their part in the process. If you don't give them all the details, they'll have difficulty making decisions themselves and helping effectively.

This only works if you can break up the solution for distribution. Otherwise, you'll spend more time clarifying everyone's roles than making real progress. Multiple people can collaborate on assets or code in several files fairly easily, for example, whereas they'll have a hard time working on the same text file and won't be able to join forces on a binary file at all. Making changes that build on each other is a much more complicated process and means you'll have to work together very closely. You can also discuss solutions with team members rather than getting hands-on assistance.

Just because you know the root cause, that doesn't necessarily mean you know what to do about it. Whether you're struggling to find a method to solve the bug or want to check what you have discovered, reach out to others and get their opinion (see "Ask Colleagues for Help" on [page 79](#)). If you have to make changes to an area you don't know well, get an expert's opinion or let them work directly on the bug. Call on an artist to fix art assets, a programmer to extend code, and so forth.

Test for Side Effects

Even when you follow all the right steps, completely understand the bug and its root cause, and implement an elegant solution, your program may still break. Other parts of the program may have become dependent on the bug, or its symptoms may be important data for some other function. To avoid this, you should always fix bugs as early as possible (see "Fix Early" on [page 196](#)).

Once you've implemented your solution, try to reproduce the bug (see "The Workflow" on [page 1](#)) and make sure the program still behaves the way it should. You can also ask your QA team to test a special version of the program that contains just your change. They probably know the software best and can quickly tell if you've broken anything else. Looking for unintended side effects is different from verifying your solution, so you should request it specifically. It may make sense to tell the team what you've changed, giving them a better idea of where to look and what to look out for.

To make comparison easier, implement the solution in a way that lets you switch between the old and new versions quickly, ideally at runtime. With assets, this process can be as simple as creating a copy and making sure you can load both. For code, you may be able to add a runtime or compile-time switch, but there are situations where a code modification is so fundamental to your program that it's impossible to change at runtime.

If switching isn't possible, you can run two instances of the program in parallel. You can take the same actions in both and observe the results simultaneously. However, not all types of programs or target platforms let you use this method.

25. Share What You've Learned

How? Share your bug fix and what you've learned with the rest of the team. Identify how the bug's root cause found its way into the program.

Why? Sharing your discoveries makes your whole team better at fixing bugs and helps them avoid similar mistakes in the future.

You or someone else made a mistake and introduced a bug into the program, and you've taken the responsibility of fixing it. You've invested all this time into tracking down the root cause, finding an adequate solution, and confirming that it solves the problem. Now pause to think about the bug and what you can learn from it.

Knowing how the bug made it into your program gives you two great opportunities. First, you can teach your colleagues. You can teach them how to avoid repeating the same or related mistakes and how to have an easier time fixing similar bugs in the future.

Second, you can improve your program. You can put measures in place that prevent the same or similar bugs from being introduced down the road. If that proves too difficult, you can make changes so that the bug is easier to notice if it occurs again.

Document the Fix

Now that you've fixed the bug and learned about its root cause and symptoms, it's time to preserve this information and consider how you can best share what you've learned.

As a first step, write an expressive and detailed commit message for your version control system. Describe the root cause, its symptoms, and information about the solution. Include details like concrete names of affected assets or components, links to further information, and either the bug's number or a link to it. If your commit message is concise at the beginning, you can provide all the details you want later.

Expressive commit messages can really help if anyone needs to find a change in the project's history that caused a specific bug (see "Check Version Histories" on [page 52](#)). Additionally, some team members like to

pay attention to repository changes, which means they'll be up to date on what you've fixed and how.

Provide the same information as in the commit message when updating the bug's status in your project management tool. This tool usually has a different audience from the version control system, possibly with less technical readers, so you may want to rephrase details or add more explanations. Ideally your version control system connects to the tool so commit messages automatically show up in the bug report there.

For details on more extensive bug fixes that you think will benefit several people, create a dedicated document, and store it with the rest of your project's documentation. Include enough information to help people fix similar bugs without having to reinvent your solution.

Comments in the code itself, and possibly linking to the bug report from there, can tell anyone reading or modifying that code that the section is prone to a certain kind of bug. Ideally, that will make everyone be more careful not to introduce another.

You can also share your discoveries in meetings. In closely knit teams, the daily stand-up provides a great opportunity for this. Bigger teams may have other sorts of regular meetings. If no suitable upcoming meeting exists, and you fixed a complicated bug with a significant impact, consider scheduling one.

In any kind of meeting, show where in the program's sources you found the bug's root cause, how it led to the reported symptoms, and what major challenges you encountered. If you know how and why the root cause entered the program, you can also discuss that.

IN PRACTICE: START BI-WEEKLY BUG MEETINGS

I've made it a routine for a couple of teams now to share stories about bug fixes in our regular meetings. Because the programming team comes together every second week, we've dubbed this part of our meeting the *Bug of the Fortnight* and always share the most interesting bug a team member fixed during the last 14 days. It's always fun—we encourage people to talk about their mistakes and sometimes encourage everyone to guess a bug's cause—and helps people remember bugs and coding principles better.

Find What Produced the Root Cause

So far, the goal of the debugging workflow was to identify and locate the bug's root cause, then eradicate it completely from your program. Now you should also try to figure out how that cause occurred.

You should assume that no one knowingly and willingly adds a bug to a program they're working on. When looking for reasons, you can consider two broad categories: ignorance and oversight.

Someone might have made a mistake because they really didn't know the correct way to modify the program's sources. (This includes cases where the person who introduced the bug had received faulty information.) They also might theoretically have known but temporarily forgotten, not paid enough attention, or not been aware of all the factors they should have considered.

While the result is the same (a bug), the ways to address and prevent further issues vary significantly. For ignorance, education is a useful measure. When it comes to oversight, you often must tackle workflows and processes.

Sometimes you knowingly make a mistake. When working on a tight deadline, you may need to cut corners and accept the possible issues, figuring that you can go back and address shortcuts later. The fallout of such an action is usually referred to as *technical debt*.

To nail down the cause, start with analyzing the artifacts the buggy implementation created. Study your program's history in your version control system as it pertains to the original commit that introduced the problem. What was the intention behind the change? What else changed at the same time? Did just one commit introduce the bug, or could multiple changes have contributed?

You should also study the component containing the root cause, such as the code, graphics, or hardware. If you're an expert in that type of component, you might be able to tell why the mistake happened. If not, ask someone who might know.

Refer to the documentation either for the specific change introducing the bug or for the system or feature where the root cause appeared. Comparing the documentation to the implementation may offer insights as

well. Also check for conflicting and confusing requirements: It's impossible to produce a bug-free program with flawed information.

As far as possible, investigate the circumstances surrounding the mistake. Looking at the project calendar can give you clues: The problem may have occurred near milestones or due dates, when lots of people were out of the office, or when other projects or events took precedence.

Ultimately, you can get only so far by looking at the information available to you, as no one will note that they're stressed and prone to putting bugs into their project.

Talk to people instead. Ask the person who made the mistake what they think led to it. If that person is you, that's even better because you should get a relatively quick reply. However, sometimes people don't remember or simply don't know why they made a mistake. Try asking more specific questions, like what instructions they were working from and what they knew about interdependencies. Ask about factors such as work environment and potential personal distractions.

TIP: BE MINDFUL OF PERSONAL REASONS

As humans, we also have lives outside of work, and we can't completely separate the two. People who have trouble in their personal lives—issues with their family, friends, physical health, or mental health—tend to make more mistakes at work.

Be very, very careful when considering this possibility and when asking about it. Your local laws and regulations may prohibit certain kinds of questions at work, especially those related to health. Many people also prefer not to share sensitive issues with colleagues.

Because the original developer may not have noticed all the factors leading to their mistake, you may need to talk to other people. Somebody else might have noticed distractions such as loud noises, interruptions from another team, or a colleague trying to help but actually sharing wrong information.

If the bug was a bigger issue and several different people have partial information on how it may have come into existence, run a postmortem meeting. Get everyone who knows anything about the bug and its cause into

a (virtual) room. Putting the pieces together may give you a complete picture of the root cause's origin.

Prevent the Bug from Happening Again

Understanding the bug's cause and how it appeared lets you add safeguards to your program that prevent similar mistakes from happening again. Take this opportunity to improve your software and put it on a better path for future development.

Evaluate anything you've added during the debugging process to see whether it can prevent bugs from being introduced or alert you to their presence. Safety and reliability can oppose other aspects of a program, like performance, usability, or even functionality. Strive for a healthy balance.

Adding safeguards to the code means you've introduced elements like assertions, error handling, or debugging tools (see "Instrument the Program to Report Problems" on [page 62](#) and "Build Your Own Tools" on [page 146](#)). Such features can be tremendously helpful in finding out more about a bug and connecting it to its root cause. Evaluate each of the code fragments you've added and decide whether you should just leave it in, tweak it so you can retain it, or revert the change.

Base your decision on the change's usefulness and impact. Keep simple checks that prevent crashes and display actionable error messages, because they help the user and don't hurt performance. If the program displays repeated warnings that slow it down and the user can't stop them, they'll quickly become annoyed.

An alternative to leaving safeguards in or removing them completely is to implement the option to switch them on or off. It's most convenient to set this option at startup or program launch. Doing so, however, might require the program to check the safeguard setting every time a safeguard is going to be executed, which in itself could hinder normal operations.

If you can't use runtime-configurable safeguards, look into compile-time options. In compiled languages, you can include or exclude some of the code based on the options you provide for the compilation. In C and C++, use the `#if` and `#ifdef` directives to test relatively complicated conditions and influence what code ultimately compiles.

Add an Automated Test

If you've implemented an automated reproduction while fixing the bug (see "Automate the Reproduction" on [page 29](#)), consider turning it into a test case for your program. Either a test that runs automatically or one that developers regularly run helps detect any regressions of the bug, because such a test treats the program as a black box rather than relying on its own reports.

In order for this sort of test to work, isolate the reproduction as much as possible. Move hardcoded repro functions to separate test modules and store repro scripts or serialized program states in a dedicated location.

Building up a library of tests that stem from your previously fixed bugs gives you a solid foundation to avoid reintroducing those problems. With a reliable process for running the tests and detecting such regressions, you can prevent most bugs from reoccurring. Turning an automated reproduction into a test case requires extra work, but it'll be worth it in the long run, so you should make it a habit whenever you can.

Implement More Relevant Safeguards

Understanding what led to your bug's root cause gives you the opportunity to add more specific and relevant safeguards than the ones you came up with when still investigating. Ideally, they should prevent the same or similar mistakes in your program's code. When you can't implement total prevention, consider alerts like turning the screen red, starting a modal error dialog, or completely stopping the program so the developer who just added the bug understands the severity of the situation.

If the bug you've fixed originated with an interface or protocol that enables mistakes, consider changing the specification. While that may require more work upfront, it'll save you from having to fix bugs later, when adjusting implementations is more difficult.

Start changing any workflows or processes that contributed to the bug. While no process will completely prevent problems, especially because people may ignore it or follow it imperfectly, you can still minimize your risk. For example, code reviews and branching workflows help reduce the frequency of problems.

Likewise, if guidelines or requirements could have kept the bug from occurring, start creating those, or adjust existing ones. These include coding standards and guidelines for asset creation.

With coding standards, you're trying to make all the code for one project as uniform as possible, so bugs are less likely. The code's style should look the same regardless of who wrote it. A coding standard ideally also establishes best practices for error handling and how defensive the code should be when it comes to potentially unsafe operations.

Approach asset creation similarly. While the asset data will be unique to each asset (no two images should look alike), their metadata and creation should be as uniform as possible. The team should establish standards and best practices: For example, different images for comparable use cases should have the same resolution.

26. Search for Related Problems

How? Once you've fixed your bug's root cause, search the code or examine your version control system to find similar problems.

Why? Bugs may be conceptually or literally duplicated in various places in your program. It'll cost you less effort to fix all of them when you've just handled one than if you have to go through the whole debugging process again later.

After you've successfully fixed a bug by eradicating its root cause, look for similar issues and fix those as well. These may be either exactly the same or so close that you can confidently identify them as related and fix them using the same approach.

While the problems you solve using this strategy may not have led to any reported bugs at the time, fixing them can prevent future issues. It's also possible the bug you just addressed obscured others, or that they just haven't been triggered yet.

You can save time for the whole team by fixing bugs preemptively. That way, the QA team doesn't have to identify, reproduce, and report them, and nobody has to figure out who's responsible for a bug or who should fix it. Also, those bugs don't waste anyone's time by interrupting their testing or use of the program.

Understand Patterns and Dependencies

Just like they contribute to bugs in the first place, ignorance and oversight can cause multiple instances of the same bug or a similar one throughout a program. Understanding patterns and dependencies can help you track down all of these instances.

One way the same bug can make its way into your program more than once is that developers add it in different places (both literally and conceptually).

A literal duplication means someone copied code from one place and pasted it in another because they thought the code was sound and that it fulfilled all the requirements for successfully doing its job in the place they put it. Because the code looks the same from a layout perspective and features the same logic, you can relatively easily find duplicate blocks by searching all your source files.

When one or more programmers apply the same problematic pattern repeatedly and in different places, you end up with a conceptually identical root cause, though the code won't precisely match. Because the code is not a one-to-one copy, a conceptual repetition of a bug is harder to find than a literal duplication.

If you've identified the origin of your bug's root cause, you can potentially use this knowledge to track down more instances. One developer may misunderstand how a technique works and apply their erroneous approach every time the requirements call for it (see "Share What You've Learned" on [page 186](#)). If that developer is a respected and experienced colleague, others will follow their lead, potentially convincing them that they're correct. The original source of the misconception might share their approach in meetings or spontaneous discussions or, in the worst case, explicitly tell people to follow their technique. Developers also like to take inspiration from others, whether consciously or unconsciously: Someone who's working on a codebase with errors may inadvertently duplicate them because they have that code on their mind when adding their own.

The same root cause can also occur by pure coincidence. If a given technology is difficult or an interface encourages misuse, people may independently introduce the same error. While the effect is the same—bug repetition—identifying the root cause becomes somewhat tricky because the problems won't share many style or logic similarities.

A change to an underlying dependency, such as a low-level system, interface, or protocol, can require changes in all dependent systems and features. If a dependency has more than one of these and not all get updated at the same time, you can end up with the same or similar bug in multiple places.

The underlying change can come either from one of your internal developers or external software that the developer modifies after you've incorporated it into your program. When you're working on an internal dependency and have to make changes elsewhere, make sure to check any similar dependencies as well. Always have a process in place for updating dependencies from external developers and run all updated programs by your QA team. If they find bugs, address them at the same time as the change.

An update may require you to modify requirements for interaction with the underlying dependency. For example, it might alter the specifications for an asset type or the pre- or post-conditions for a function. You have to implement updates in response to the underlying change in all assets or places in the code that the original affects.

Ideally, the program alerts the developer instead of silently failing when it hasn't gotten a dependency update. Each asset or protocol message should contain a number specifying the version of the data it contains, which makes it easy for the system that reads the data to determine whether it's compatible. If it's not, the system can provide a specific error message and tell you what the best course of action is.

When it comes to code changes, the compilation should fail if an underlying system was changed and the code isn't using it right, which draws your attention to all the areas that you'll need to modify, so you'll quickly know what you need to update and how.

Avoid revisions to underlying systems that the compiler accepts without failing. Here are a few examples that are difficult to find:

- Changing the meaning of a function parameter without changing its type
- Changing the type and meaning of a function parameter, but keeping the types compatible, like an `int` being converted to a `bool`

- Changing the pre-conditions of a function without verifying them as the very first step in the function

IN PRACTICE: UPDATE ALL PLATFORM-SPECIFIC CODE IN ONE GO

In a few projects, I needed to port a successful program to a new platform with the goal of having it run in precisely the same way. A platform tends to have a significant impact on the code because each has its own SDK. While most code runs on almost any platform nowadays, you always need to account for minor differences.

However, most of the work came from one of the projects having a lot of platform-specific code that was wrapped in `#ifdef PLATFORM ... #endif` conditions. In many cases, it was productive to address all `#ifdefs` in a certain system at once because they made it conveniently obvious which parts of the code I had to adjust for the new platform.

Make Use of History

The history in your version control system (see “Check Version Histories” on [page 52](#)) can help you in your search for similar problems. As you’ve already located the bug’s root cause, you shouldn’t have trouble finding the corresponding commit. Most version control systems offer a *blame* function that tells you which commit modified each line last. Looking at the history of the individual files also makes it easy to determine which commit added the problematic change.

Once you find the commit, first check what other modifications it contains. It may add comparable errors in other program locations. Take a close look at each change the commit makes and compare them to the root cause. They might look very different but have similar effects.

Next, look at the commit’s description in the version control system and try to understand the gist of it. In an ideal world, the commit message contains a high-level description of its effect. If you have that information and search the rest of the project’s history for similar intended effects, you might be able to find other commits that introduce similar bugs. For example, if one commit message says, “adds undo support to the text box feature,” and the bug was connected to your undo system, another commit adding undo support to other features may very well be problematic.

By finding out which commit introduced the problem, you've also discovered who made the mistake. If the other lines of investigation didn't yield any results, you should check whether the author potentially added the same bug in other places. Filter the project's history by author and check the commits for related messages and source changes.

Consider comments in your project management tool as well. You might be able to spot related tasks or bug reports from the commit description or notes in the code. If your organization's standards reference a task for each commit, read up on the related task. It might mention other places with the same changes. Others may have reported similar issues in the comments as well.

Finally, check whether duplicate files in your project appear in different locations. A feature may need a file to exist in several places. If that file relates to the bug, you may need to update all those instances.

Search the Code

If a code change introduced the root cause, search the whole project for fragments of the code in question. Use fragments that don't contain any concrete data like text or numbers and that are generic enough to be present in other systems but not so generic that they appear all over the place.

Consider this C# code:

```
int[] numbers = {1, 2, 3, 4, 5};  
for (int i = 0; i <= numbers.Length; i++)  
{  
    --snip--  
}
```

Searching for `<=` will give you way too many results, while `{1, 2, 3, 4}` might not give you any. However, searching for `int i = 0; i <=` might give some good results because it should return all spots with a `for` loop that makes a lower-or-equal comparison.

It also makes sense to search for similar comments. Again, don't look for concrete data or names but for fragments likely to come up again.

Shared references can provide you with another lead. Look for references to a dependency, such as another program, an SDK, or a

framework. Slightly more distant dependencies include assets, remote server addresses, or the names of other programs yours communicates with.

In the case of a code dependency, examine the function and system your code is calling into. Then search for their other call sites. Alternatively, many IDEs can find all references to a specific function, which makes the search a lot more effective. With object-oriented programming, it can be difficult to find all function calls just through a text search.

Evaluate each call site you find and compare it against the root cause you've fixed. Does the function call make the same assumptions and possibly fail in the same way? Can you apply the same fix? Debugging the program as it executes helps a lot in this case (see “Step Through the Code” on [page 100](#)) because it allows you to compare the runtime decisions and the data between several potential instances of the root cause.

One more way to find similar causes is to investigate the whole system in which you fixed the original. There are two reasons for this. First, a system may have to make use of similar logic in several places, and because the author is usually one person or a small group, they tend to express the logic in similar ways. Second, the whole system might turn out to be suspicious because it has already contained bugs multiple times (see “Try Shortcuts” on [page 38](#)).

Scan the system manually for similar code. Watch out for similarly named types, functions, variables, and comments. Even the shape of the code may give you a hint. If the indentations or control-flow statements are comparable, the logic might be too. As before, if you've found a possible match, double check to make sure it's correct.

Search the code for references to external dependencies' identifiers, like a server's hostname or URL, an asset's name, or the path pointing to it. If you're lucky, there's really only one correct way to reference a dependency, and you can find all occurrences with a single search term.

Make Bulk Changes with Caution

When making changes based on similarity to a root cause you've already fixed, you must be very careful. You're not solving a problem that already exists; you're trying to prevent issues from arising later—which might never happen. If you make a mistake when modifying the code and break something, others may think you added a new bug for no good reason.

You're also circumventing a large part of the debugging workflow if you fix a bug without properly investigating it. When you know where a mistake might lie, at least try to reproduce your bug, which should be easier since you know the root cause and how it manifested elsewhere. This way you can verify that your change fixes the bug even in a different spot. Your QA team can help with reproduction and verify the solution.

IN PRACTICE: HOW DID THIS EVER WORK?

When looking for duplicated causes, I too often find a code problem and wonder "How did this ever work?" Of course, there's always a logical explanation for it, but lots of issues initially leave me stumped.

The reason is usually trivial. Upon further investigation, it may turn out that a particular piece of code hasn't been executed for years. Lots of people may have observed a problem but thought it was intentional. In the worst case, several people saw the bug but assumed it had surely been reported already because it was so bad.

Don't look too broadly for similar instances. You must be convinced the pattern really is the same and that you've found a bug that will cause problems later. When making any speculative changes, make sure you understand each individual spot and verify your fix in each one. Only then can you say for sure that you fixed the same issue and commit the change to your version control system.

Share the details of your changes with your QA team so they can focus on testing those areas, though they should continue with their program-wide trial to make sure you haven't broken anything critical elsewhere. The challenge for the testers in this case is that they don't have a bug report with reproduction steps they can use to verify the fix.

Finding similar bugs is also an excellent opportunity for a code review. Make sure your colleagues completely review all the spots you've changed. Tell them to use your solution for the original root cause as a template for your modifications elsewhere.

27. Fix Early

How? Fix any bugs as soon as your schedule allows.

Why? If you've recently worked on anything related to a newly reported bug, you'll have an easier time correcting it when your memory is fresh. Additionally, the longer a bug remains in your program, the more functionality will start depending on it, which makes the removal harder.

When you can start fixing a bug depends on your development process, the bug's severity, and when you last actively worked on the system that contains it. When testing and iterating on what you've implemented, testers and developers identify bugs and developers address them. Once a project or component is done, minor issues drop to the bottom of the agenda, though critical issues still get high priority.

Assign Reasonable Priorities

Ideally, you would fix each and every bug immediately after someone reports it, but in reality, other work usually has a higher priority at that moment. That's why your team needs a bug-handling procedure that describes how to report bugs and determine their priority, as well as who starts working on them when.

Critical bugs that block users or other developers take precedence over everything else, but you must know how to schedule more minor issues. If you're working on a different task, an interruption could delay you by making you switch contexts and potentially your work setup. When determining a bug's priority, consider not only the severity but also the effects of long-lasting bugs.

The bug's presence might affect your product's and team's reputation. It's difficult to sell a high-quality product that contains annoying bugs, and users' bug reports and complaints can take a lot of effort to handle.

Bugs could also slow down testers and developers if they affect the program's usability or reliability. QA testers will run into the same bug repeatedly if it stays around for longer. If they're not careful, they may also report duplicates. Additionally, if a bug stays in your program for an extended period of time, parts of the program will likely become dependent on it and then break when you fix it.

Leverage Context

Fixing a bug's root cause requires a lot of knowledge about the environment the bug exists in and how exactly the cause connects to the symptoms, as [Part II](#) and [Part III](#) show. If you've recently worked in that environment, finding the connections becomes much easier. You know the general architecture of the program, how the individual components and systems interact, and what sources contribute to which feature. You're also still familiar with the development workflow, version control system, tools, and experts on your team, as well as the building, packaging, and deployment processes. Finally, you have everything set up on your development machine, a checkout of the repository on your local disk, and access to any dedicated or rare devices you're targeting.

You may even be able to guess the cause just from the description of the bug's behavior (see "Try Shortcuts" on [page 38](#)). Even if you need to make more significant changes or refactor the code extensively, a general understanding of the context will help.

A bug assigned to you might turn out to be more of a misunderstanding. In such a case, tackling the problem quickly makes you more able to clear up the confusion. (You should also tweak the implementation to avoid further issues.)

IN PRACTICE: KEEP PROJECTS SMALL TO STAY FLEXIBLE

A little while ago, my team was working on a number of technically similar projects that each took a couple of months.

We could share a lot of code by copying and pasting. Because we weren't maintaining the projects for an extended period, we didn't start merging the codebases to reduce duplication.

However, we repeatedly had to test one of the projects while working on another. Due to the nature of shared code (and assets in some cases), we also needed to copy code fragments and assets between the projects regularly.

The projects' sizes complicated all of this. They each came in at around 200GB without build artifacts, and 225 after building. Our main SSD at the time only had a 500 GB capacity, so we couldn't fit more than two projects, which made us less able to work on two at once. Some colleagues started using their HDDs for development, which had a larger capacity but also increased build times severely. Others copied projects back and forth between the SSD and HDD as needed.

In hindsight, we should have looked into reducing the projects' sizes to the bare minimum by removing any unneeded data or purchasing bigger SSDs. For more long-term projects, we should have identified shared code and data, so we didn't have to maintain redundant copies.

Prepare for Attrition

Realistically, you have to expect people to leave your project, team, or company, taking their context with them. Even a reassignment means a person has less time and focus to deal with bugs from previous projects, while someone leaving the company will take their knowledge with them.

Always try to have multiple people work on essential systems so they automatically share knowledge. If that's not possible, at least make sure to keep the whole team informed about systems' internal operations. You can do so through asynchronous communication like code reviews or documentation, as long as everyone reads the information they receive.

Incorporating discussions about architectural decisions and implementation details into your daily stand-ups can spread knowledge. You can also establish regular meetings to discuss the latest developments, or schedule sessions on demand when you reach a bigger milestone.

Rotating responsibilities is another option. Have one person work on one system for a month or a week and then switch them out for one of their colleagues. Work will be a little slower while the developers get up to speed, but this process will force them to really understand their systems and communicate the little details that are hard to get from the implementation alone.

TIP: DISCUSS THE BUS FACTOR

The *bus factor* refers to the number of key people within a project or team whose absence due to unforeseen events (like getting hit by a bus) would significantly disrupt or halt its progress. A low bus factor implies heavy reliance on a few people, posing a risk to the project's continuity. While the concept is a bit tongue-in-cheek, it usually gets a good discussion rolling regarding knowledge concentration. Everyone knows they won't be staying at the same job forever.

It's easier to share the high-level work that influences key implementation decisions. Involve the whole team or at least several members in the design and architecture of new systems and larger refactorings. If you know what the goal of a particular system is and how it

functions in theory, that's already worth a lot and will help you in case you ever have to investigate bugs in it.

As soon as you know that someone will leave the team, start conducting extensive handovers. Review their existing documentation and figure out who knows what about their expert areas. Next, make a list of topics that have to be documented. Ask both the team and the person who will leave to add anything to the list, then ask the departing employee to address all the topics in writing. Check the results against the list.

Also make sure employees have opportunities for verbal handovers. Assign ownership of each topic to a specific person and have them discuss it in depth with the employee who's leaving.

Avoid Dependent Behavior

Dependent behavior happens when part of a program works correctly only as long as a bug is present and exhibiting its faulty behavior. The dependency may be compensating for that behavior or taking it into account as ground truth. As soon as you have such dependencies, you'll have to fix them as well as the bug.

Some dependent behaviors are easier to deal with than others. Fixing a bug's root cause can fix anything involving runtime logic. You'll have to repair assets separately, though, if they work only because of the bug's existence. Even worse, if the bug has an effect on users' saved data, you'll have to repair that even after fixing the bug.

Distinguish carefully between *transient data*, which is generated and consumed during one execution, and *persistent data*, which is stored between executions. Changing any logic related to persistent data probably requires a compatibility or upgrade path, which can be either integrated into the program or a dedicated outside tool.

For example, if you change the way a text file format expresses links, updating all the documents you've already created will be quite a chore. If you offer the format to users, you should definitely write a conversion tool for them.

IN PRACTICE: REMOVE A LAYER TO FIX A BUG

One project I worked on dealt with input devices that needed dead zones to function correctly. A *dead zone* is a range you define in which the input signal is treated as absent. This avoids accidental activation due to imprecise hardware.

Over the years, this project had acquired two dead zone implementations layered on top of each other, processing the input signal twice in a row. A colleague unsuccessfully tried to correct this over a few days, then figured out they simply had to remove one of the dead zone implementations because it conflicted with the other. After that, they fixed the bug very quickly.

Another form of dependency is when your users start liking the effects of a bug that you want to remove from the program. (For example, the combos in the fighting game *Street Fighter II* were originally the result of a bug, but they made such a great addition they turned into a feature and became genre-defining.) So why not keep it? Simply retaining a bug as a happy accident always poses a risk. The bug might make the product as a whole worse or less profitable. Because it's unintentional, it's also unstable, so even changing unrelated parts of the program might have unpredictable effects. If you've stumbled upon a useful feature through a bug, you should certainly go back and implement it properly (see “Declare It a Feature” on [page 213](#)).

Take Ownership

If you've developed the systems the bug affects or contributed to them extensively, you should assist with prioritization even if you won't fix the bug yourself. You can provide helpful context, a time estimate, and ideas and pointers on how to solve the problem.

You're also the best person to handle bugs in *your* systems. Keep an eye open for any relevant bug reports in your project management tool—and an ear open for complaints. The first indication of a bug is often people talking about the problem they just ran into.

If there's a bug you learn about by being attentive to your surroundings and environment, tell the person who found the bug that you'll handle it. As soon as it's properly reported, assign it to yourself or reach out to the person responsible for distributing bugs and tell them this one is yours.

One more way to ensure bugs are handled promptly is to reserve a fixed portion of your work time for bug fixing. Otherwise, new

developments or other work responsibilities may prevent you from working on bugs, or you may forget to look for them. You may want to create an appointment in your calendar so others can tell you're unavailable. Squeeze as many bugs as possible into this reserved time. If you should run out of bugs, that's a great problem to have, and you can easily go back to implementing new features. Make sure to work in order of bugs' priority.

Your colleagues will appreciate your proactivity if you look for opportunities to find and remove bugs. Fixing bugs early also allows you to avoid building up technical debt.

28. Deploy a Quick Fix

How? If the bug was recently introduced, you can revert the offending change using the history in your version control system. As an alternative, you can implement a workaround that masks the bug's symptoms.

Why? Both approaches lessen the bug's impact and allow users to continue running the program without a proper bug fix in place.

To provide a better experience for anyone using your program and unblock anyone affected by the bug, publish a solution as soon as possible. While fixing bugs in your program properly is essential, it's also crucial not to waste other people's time. Implement a temporary solution first and share it if you expect the proper fix to take longer. Then you have more time to continue the search for the correct remedy.

Undo the Offending Change

One method to fix a bug temporarily is to identify and undo the change that introduced it. While there certainly was a reason for the modification, it can probably take a back seat for a little while. Once you've found a solution, you can reintroduce the change along with it. You can either revert the entire change and restore the files to their former state or manually revert any related actions.

Your team may not be responsible for some buggy changes. For instance, if a dependency is responsible, you could try reverting it, but it

might be out of your control. The more recently the bug entered your program, the easier it'll be to find the offending change and remove it.

Use Your Version Control System

Your version control system can undo any change previously committed to the repository once you identify the responsible commit. While this is most straightforward if the bug appeared recently (because you either know the commit already or can rule most commits out since they don't involve the related area), it's still possible with changes that happened a while ago. See “Check Version Histories” on [page 52](#) for an approach.

The next step is to simply revert that one change in your version control system. You may have to do this by hand, going through all affected files and applying the opposite modification. However, most version control systems have an explicit command to handle that kind of task. For example, Git's `git revert` does exactly the opposite of a specified commit.

If you can't cleanly revert the commit, you have to manually adjust the reverting change so you can implement it without introducing any new bugs or build issues. For example, if a code change doesn't apply cleanly, it likely breaks the syntax and prevents compilation. Even if the syntax works out, the logic may be wrong.

This way of undoing a commit is often used on modifications a colleague made before they left for the day, ended the week, or even went on vacation. They may have wanted to finish their work so nobody had to wait too long for their changes, so they committed them and made sure they didn't break the build—admirable in theory, but once they left, they couldn't fix any bugs that appeared later.

Depending on the complexity of the change, understanding the commit and finding the bug can take a lot of effort. It's usually much more productive to undo it and wait for your colleague to return. Only try fixing the bug yourself if the change is critical.

IN PRACTICE: UNDO A CATASTROPHIC CHANGE QUICKLY

A while ago, a colleague on a project was working on a tool we all used to convert assets to different formats. The colleague implemented a new feature that cleaned a target folder of all files, built the tool, and shared it. After we used it for a while, people noticed that some of their files were missing. We discovered that if the tool wasn't in

the right place, it would delete all files in the current user's *Documents* folder instead of the ones in the target folder.

The only reasonable option in this case was to undo the committed change, build a new version of the tool, and distribute it. While some files were lost, our swift action (as well as warning the team) prevented the worst.

Revert Manually

Undoing a change manually potentially allows you to be more selective about what parts you revert. Knowing what the modification did, you can go into your program's sources and do the opposite. If the offending change added something new, such as a file, feature, or dialog, it should be straightforward to remove it again.

When it comes to source code, delete the file or code fragment. You can also comment out the added code or, if it's self-contained, comment out any calls to the offending function. You may have to patch dependencies or references from elsewhere to make the code compile.

Reverting data or an asset can be as simple as removing the file or any references to it. If data is part of a bigger document, you may need to undo the change there. This is doable in text-based formats but much harder in binary. While you can theoretically use a hex editor to modify binary files (see "Use Off-the-Shelf Tools" on [page 164](#)), you'll likely end up corrupting them. Instead, replace binary files with older, working versions.

Once you've reverted any changes to your program's sources, make sure it works as expected and that the modifications are actually gone. When fixing up the program afterward, be careful not to do too much to make it work, as that may accidentally undo what you've reverted. For example, when commenting out a fragment of code, it may be necessary to adjust the code around it to make it compile. You may be tempted to bring some of the commented-out code back in, but that could leave the code in a state where it's unclear to what extent the change was reverted.

Implement a Workaround

Another way to provide a quick fix is to implement a workaround, a patch that circumvents the buggy behavior but doesn't offer a sustainable fix. It should either prevent the program from triggering bug symptoms or prevent those symptoms from reaching the user. Therefore, you'll need to

understand where and how the bug causes problems in your program if you're going to implement a good workaround.

Once you've set up the patch, you have to test your program and ensure it's safe to use again. If your bug resulted in a crash, the program should display an error message instead, for example. The workaround might also prevent users from taking actions that trigger the bug, like opening a window that can cause visual corruption.

Masking symptoms with a workaround can easily backfire. The user may be confused and file a bug report because they can't take a specific action at a given time or they get a cryptic error message when they do everything correctly. New functionality might start depending on the workaround and build upon it, making it difficult to remove (see "Fix Early" on [page 196](#)). That said, implementing a workaround can give you your first insights into what makes the bug tick and put you on the path to finding the root cause.

Alter the Code

Depending on where the bug's symptoms are in your program's sources, you may have to change the code to hide or circumvent them. Always add descriptive comments so anyone looking at your workaround understands its purpose.

The best method of changing code depends on the individual bug. You could tweak conditions to always choose the case that doesn't cause the bug, change thresholds or timeouts so an operation doesn't fail, and hardcode variables to reflect the correct values even if the program can't produce them at runtime (see "Experiment" on [page 134](#)). For example, you may want to always refer to a specific asset for a given use case to ignore a broken one that the code in another area or a different asset points to.

In another approach, you choose a different trade-off than you usually need. Making the code simpler trades stability for performance or usability: Simpler code gives rise to fewer mistakes but may not be as optimized or take all edge cases into account.

Some bugs may appear only after the compiler optimizes code. If that's the case for yours, try disabling optimizations for the code section most likely to contain the bug's root cause. You'll have a compiler-dependent

way to disable optimizations. For example, use `#pragma optimize("", off)` for Visual Studio or `__attribute__((optnone))` for Clang.

The best tactic is to write code that's both simple and defensive. For example, if the bug is connected to finding an element in a container, use a linear search through the container instead of a hash table. If multithreading is involved, disable it entirely and run all operations in a serial fashion instead. (Adding lots of mutexes, even in places where you think they should be unnecessary, can make this easier).

In case a specific system or function causes the symptoms by reporting an error, try ignoring that error and continuing as if nothing had happened. Comment out assertions or return value checks completely. Hardcode the offending function to always report success.

When it comes to exceptions, you can comment out the statement throwing them. Alternatively, you can isolate a whole area from exception-related issues by placing a `try-catch` block around it and then not reacting to the error (or just printing a warning). This easily keeps your program from exiting due to an unhandled exception.

One more brute-force method to hide a bug's impact is to automatically restart your program. While you may not be able to do this completely under the radar, it still prevents error messages from bothering the user and means they don't have to restart manually. Restarting wipes the program's slate clean, solving all issues caused by knock-on effects from an earlier bug, unless the bug has modified the environment and not just the program.

If you can take the user back to precisely where they left off and the restart doesn't take too long, they may not even notice it. Some programs lend themselves better to this approach than others. Restarting an application in which the user's activity is going to be obvious, but a backend process that serves self-contained requests every few seconds is another story.

TIP: RESTART DURING THE LOADING SCREEN

Restarting programs without the user noticing may sound far-fetched, but it happens! *Morrowind* would restart the original Xbox during some loading screens because it would run out of memory otherwise, and the users had no idea. How? The developer integrated the game and the system so tightly that the loading screen would stay up during the whole restart, and the game would pick up exactly where it left off afterward.

Modify the Assets

You can also modify one or more assets to mask bug symptoms. If the code references the wrong assets or parses them incorrectly, you can adjust the assets to compensate. If the asset itself causes the problem, the quick-and-dirty fix is to replace it with a working asset of the same type that has the same properties and isn't so radically different that the replacement is obvious. For example, if you need to replace an image, the one you put in place should have the same resolution or at least the same aspect ratio. (Also, be sure to put it in the right spot.) If you need to build assets, you can make the change permanent by replacing the source file, but you can also modify the target to quickly test the workaround. Example assets from sample projects make great replacements, if you have one that's appropriate, because the person who supplied them knows a lot about what works and what doesn't.

Alternatively, you can use a minimal asset: an asset that technically works but contains only the minimal information to be valid. Examples include empty text files, JSON documents that contain only empty objects (`{}`), or image files that contain 1-pixel black squares. Minimal assets don't take much effort to create or use advanced features, so they may not touch on any complicated parsing logic and will load faster while using less memory.

Change the Environment

The final way to work around a bug is to change your program's environment— although this works only if the environment influences the bug. Some problems arise from technical specifications or limitations, such as not enough memory, while others show up because a program is incompatible with components like CPUs or USB devices. In these cases, you can switch your technical specifications, add components like memory or a faster graphics card, or use a completely different machine.

Any program that runs in parallel to yours on the same device or runs on another device it communicates with may cause problems. If the other program is not crucial to yours, try removing it. If your program does

depend on the other one, try switching the latter to a version compatible with yours. This even applies to the device's operating system.

The physical conditions of the device your program runs on are also part of the environment. For example, your program may react to changes in the gyroscope or acceleration sensors, showing buggy behavior that goes away when putting the device in a specific position, moving it, or holding it still. Another relatively common issue occurs when your program uses a lot of the device's resources, making it run hot, and the built-in temperature sensor throttles the CPU in response, slowing the program down. Adding a cooling fan or turning on the AC may prevent the problem.

Communicate Workarounds

You may know how to work around a bug as a user, but not how to translate that into code, assets, or the environment. For example, Microsoft Excel doesn't let you have two files with the same name open simultaneously. As the user, you can get around that by renaming them.

If you know of a procedure to keep the bug from occurring or its symptoms from manifesting, you should share that knowledge with its users. A manual workaround will let them use your program again and keep them happier while you try to find a solution.

Document the workaround well so it's easy to follow. Advertise that you have it, where to find the instructions, and what the effect is. Ideally, talk to affected users directly and help them learn about the workaround.

Don't Stop at the Workaround

Once you have a hacky fix, publish a new version of your program for users while you keep looking for a proper solution. Let others know about it so they can hold you accountable. When you add the change to your version control system, state explicitly in the commit message that it contains a workaround, temporary solution that needs to be redone. Add finding a real solution to your project management tool as a task and assign it to yourself.

Mark any temporary code changes with a comment containing a big `FIXME`, your name, and a description of what you've done and why. If you need a new file, give it a name that denotes its temporary status.

Even if you've found what seems to be a reliable workaround for your bug, keep looking for a permanent solution. Otherwise, you'll be building

the rest of your program on a shaky foundation that makes future bugs more likely. The band-aid you just added isn't going to hold for long.

You may have to perform more iterations of the debugging workflow, which will cost you time. However, you put the temporary fix in place to give you just that. Your goal should always be to fix the bug properly (see “Just Fix It” on [page 182](#)).

29. Start Over

How? Rewrite the functionality that contains the bug. Once you have a working version, you can compare the two implementations to help identify the root cause in the original.

Why? The rewritten functionality likely doesn't contain the same bug the original implementation featured because it was written at a different time, with different requirements, and ideally by a different person.

If you're working on a particularly mean bug or in a very complex environment, you might find it easier to reimplement the functionality from scratch than to find the bug's root cause or a working solution. While this may seem like overkill at first glance, it can be a viable option for systems with recurring issues or unnecessary complexity, which you should simplify anyway. Rewriting a functionality gets rid of both the bugs you know of and the ones you don't—as long as you're sure it contains their root causes.

Just be careful when deciding on a rewrite. It's easy to underestimate the effort it will take or assume you know everything about the system, but you probably haven't encountered all the little tweaks that make a system work in the real world, particularly if it's been running for a long time. Such tweaks are usually not part of a system's specification or initial design and are ultimately not documented anywhere. Rewriting functionality is more realistic on a smaller scale, such as for individual functions or small sets.

Rewrite the Offending Functionality

Scrap the functionality that presumably contains the bug's root cause and rewrite it from scratch. It's essential that you don't just refactor it or leave the overall structure intact while changing only smaller pieces here and

there. Instead, ensure you understand the functionality's purpose and reproduce that with a completely new implementation.

Try a more straightforward approach this time, avoiding potentially tricky details like multithreading, raw pointers, and direct memory access. For the first step, it doesn't matter if the code doesn't fulfill all your regular quality requirements. You can take care of performance, maintainability, portability, and other details after you test on the target device under the circumstances described in the bug report.

While you can ignore quality requirements that extend code life, you should follow all best practices to make the code reliable, clean, and easy to understand. Play it safe by using only algorithms, techniques, containers, and so on that you know well and trust not to have produced the bug.

Additionally, write unit tests to cover all the individual pieces of your implementation and integration tests for the functionality as a whole. Ideally, those tests cover the initial failure case that led to the bug, giving you a good starting point to test yours against. Rewriting functionality can also be a good scenario for using test-driven development: writing the tests first based on the specification, then writing the implementation against the tests.

To make the implementation easier and therefore less error-prone, add functionality and features iteratively in small increments. Compose the functionality from small, testable pieces. This way, it's easier to write each individual increment and review it yourself to rule out any potential issues. You'll also be less likely to make mistakes in the process. Evaluate each increment thoroughly using unit tests and by using the functionality as part of your program. The goal is to detect problems soon after adding them to the code.

Try developing the reimplemented functionality outside of the main program. Of course, that's not possible in all cases, but it's a useful way to speed up iteration times and rule out interaction with other systems as a source of problems. This method also forces you to write a more self-reliant version. You may have to provide stub implementations (see "Stub the Implementation" on [page 212](#)) for other functionalities, which is a great way to identify actual dependencies, but you should keep these stubs as simple as possible.

Even when developing a new implementation independently, ensure it will fit right in. Use the same SDKs and the correct interface from the start so you can plug your implementation into the rest of the program later. Otherwise, you may end up with an implementation that you can't test against the original and won't be able to find out whether you've fixed the bug.

It's also helpful to involve other people from your team (see “Ask Colleagues for Help” on [page 79](#)). Generally, the developer who wrote the original implementation shouldn't handle the new version, though, since a different person is less likely to make the same mistake. Teaming up can also make writing lots of code easier. For extraordinarily tricky parts of the implementation, try pair programming, where you sit in front of the code together and write it with at least two pairs of eyes to catch errors.

Once you're finished, ensure that someone reviews your code thoroughly. Tell the reviewer assigned to your implementation that you're trying to fix an important issue.

While you can't use all these precautions and extra processes while developing the whole program, they can help you a lot in a situation that demands care. Sure, you may still make a mistake, but the more safety nets you put in place, the more likely one will catch it.

Use a Clean Room Implementation

You can use two fundamentally different approaches to discover what functionality you need for your reimplementations. The first is a clean room, in which you don't know anything about the original implementation. All you have is the functionality's documentation, a complete specification of all requirements. That includes the following:

List of valid inputs This list includes input types and data ranges. Anything that comes in from the outside and affects the functionality, such as data from a file, global variables, and thermometer readings, counts as an input.

List of valid outputs This list shows both types and data ranges for anything that leaves the function and has an effect on the outside. For example, data written to a file, return values, and logs count as outputs.

List of possible failure cases This describes what happens if the functionality receives inputs other than the valid ones, precisely specifying the effect of each invalid input. The program might crash, throw an exception, print an error message to the log, or return an error value.

Derivation specification This tells you how the functionality operates and what the outcome is. The description could reference an algorithm, but it might also have to go into more detail about what is supposed to happen.

Of course, the whole specification must describe the functionality's intended and expected operation. It mustn't describe the one including the bug because that would cause you to reimplement it.

Such a specification is rarely readily available for any program aside from safety-critical systems. You may find bits and pieces in some of the related documentation. Look for design documents, implementation specifications, and diagrams, all of which may hold some precious information. If you can't find any of those, documentation targeted at users or other developers can give you an idea of the intended functionality.

IN PRACTICE: REIMPLEMENT BASED ON PROTOCOL SPECIFICATION

In the early days of my programming career, I thought it was a great idea to implement a bespoke HTTP server for a web service we were developing. The server worked fine during our tests, but when we went live, it had trouble parsing HTTP headers in some situations. Because I couldn't figure out what those situations were, I chose to reimplement HTTP request handling on the server side.

Based on the HTTP specification, I rewrote the way the program reads bytes from the incoming network connection, converting them into strings and interpreting those strings. The result worked a lot better. However, nowadays I would use an existing HTTPS server template because users and search engines both prefer HTTPS, and the encryption aspect is especially difficult to implement correctly.

You can also talk to developers who've previously worked with the functionality's code. The original developer may be able to provide a lot of information, and anyone who's changed or used the code before could also help. However, they shouldn't perform the reimplementation: As mentioned

above, they might inadvertently use the same approach and reintroduce the bug. You should also avoid referring to the existing implementation as far as possible, though that depends on the specification's availability, quality, and completeness. After all, if you don't know what the functionality is supposed to be, you can't reimplement it, and the existing version may be the only source of that information.

Use a Dirty Room Implementation

In a dirty room implementation, you analyze and read the original before starting work and keep referring to it throughout. You want to conceptually transfer the functionality to the new version.

Be careful you don't end up mindlessly porting the old implementation to the new one. First and foremost, don't just copy and paste the code from one location to another. Don't read about the original and then reproduce a very similar implementation line-by-line either. This can quickly happen if you don't consciously avoid it.

Instead, when reading the original implementation, you have to abstract from it. Understand the code's function and purpose on a high level. The more experience you have, the easier it becomes for you to identify common design patterns and well-known algorithms. Once you know the implementation's purpose, you can translate it back into a new version without copying any code.

To facilitate distance between the old and new implementations, you should keep them visually apart. Only refer to the original code if you're stuck because you don't know what functionality to implement. If you have a hard time understanding a concept in the code, learn about it rather than applying it the same way. Look it up online or talk to your colleagues. If you can't figure out a concept, don't use it. You have to understand what you write to know if there's anything wrong with the code.

A dirty room implementation becomes necessary when there's no documentation available about the functionality you're going to rewrite and there's no one around to ask about the requirements. Many development teams emphasize shipping their program instead of diligently documenting it or focusing on implementation quality. Additionally, the older the code, the less likely that anyone is available to discuss it.

You may be able to pair the reimplementation with a minimal reproducible example (see “Build a Minimal Repro” on [page 210](#)). Both should be standalone programs.

Test the New Implementation

The new implementation has probably already passed some rounds of isolated testing, but the next step is to integrate it with the main program to find out whether it does fix the bug. Try the reproduction multiple times and observe what happens. Is the bug still reproducible? If it’s not, are there any other adverse effects? In either case, you’ll have to go back and fix the reimplementation. Either you’ve made a mistake in the code, or you haven’t gotten the requirements quite right yet. To fix the code, try more tests or involve a developer with experience and an eye for bugs. You may have to consult the original implementation’s code to get the requirements right, even if you’ve tried to use a clean room so far.

If you’ve successfully fixed the bug with your reimplementation, the first question is whether it’s possible to simply keep the new version and be done. Evaluate your implementation against your team’s quality requirements. While it might be good enough if you didn’t have to make too many compromises, it will more often lack quality or features. In that case, you have to use it to get the original implementation working.

If both implementations give you identically wrong results, either the new one is broken as well or the functionality isn’t the problem. The bug may well be in another spot, with your functionality as a dependency.

Compare the Results

To determine the problem with the original implementation, compare the results with those from your new version. Slowly work your way backward to pinpoint the actual problem. Your comparison method depends on how frequently your program calls the functionality and how often and under what circumstances the faulty implementation fails. If it happens often and is easy to reproduce, you can compare the results manually. Grab the output values from the debugger or print them to the log or on the screen. This works best for text and numerical values. Bigger outputs, such as images or large arrays of data, may require you to use a tool or create a smaller output.

Provide a Way to Switch

To facilitate comparing results, provide an easy way to switch between the two implementations. Runtime switches let you compare outcomes without having to restart the program. If you can't use one, provide a way to switch on startup through a command line parameter or configuration setting. Use a compile-time switch only as a last resort because it has the longest iteration times.

The quicker you can iterate between the implementations, the more easily you can experiment with them. You may want to test in several different scenarios where you then need to toggle between versions, for example. For visual discrepancies, rapidly flipping back and forth can reveal differences or a pattern.

Parallel implementations enable you to iterate even faster and potentially see both sets of results on the screen at the same time. As well as fixing a problem more quickly this way than otherwise, you can make sure your new version produces accurate results (except for bug symptoms).

You can provide the same inputs sequentially rather than running the programs concurrently on different threads, but make sure one doesn't influence the other. While you should have ruled this out by collecting a complete list of inputs, the implementations may still share hidden dependencies or parameters, of course.

Hidden dependencies could cause one version to behave differently when you run it after the other. This gets especially tricky if the functionality transforms some data that you might accidentally reuse as the input for the second implementation. In such a scenario, you must make a copy of the data before running the first version.

Automate the Comparison

For implementations with many different inputs and rare discrepancies in outputs, it's better to automate output comparison. If you can run both versions in parallel, automation can help you spot differences you might otherwise miss.

Of course, you'll have a harder time automating some comparisons than others, such as large or complex outputs. In such cases, you have to switch between implementations at runtime after all. If you can find a

smaller part of the functionality where you're sure the root cause lies, you can also reimplement and compare that.

If your program detects a difference automatically, it should be able to show where and how the outputs diverge. It can even alert you as soon as it detects any discrepancies, which is essential if you can easily miss them. This alert can take the form of an error message in the log, a programmatic breakpoint, or an on-screen message. It must allow you to analyze exactly what happened when the program spotted the issue (see “Instrument the Program to Report Problems” on [page 62](#)).

Having found differing output values either through automated comparison or by catching them yourself, you need to dig into the input values and figure out how they relate. The discrepancies should point you toward the bug's root cause. Keep a close eye on the data and its transformations (see “Follow the Data” on [page 122](#)) and debug the code (see “Step Through the Code” on [page 100](#)).

Once you're done with the bug, don't just throw away that new implementation. You may need it if another bug shows up in the same area (see “Share What You've Learned” on [page 186](#)). Additionally, if you've implemented an automated comparison, you can run the alternative implementation as part of your test suite to proactively detect bugs.

Stub the Implementation

An alternative to a complete reimplementation is creating a stub, a trivial version that always returns the correct outputs based on a known set of inputs and refuses to work with inputs it doesn't know. You can hardcode the correct results as the output values after you've calculated them yourself, generated them with another tool, or pasted them from an earlier execution where the problem didn't show up.

For example, when trying to rule out an issue with your networking implementation, you can stub out network functions by providing implementations that only act as if they were communicating with a remote program. When they return success values, the rest of the program can continue without issue if the problem comes from networking.

You can gradually extend a stub to cover more input ranges. That way, you either run the bug again or at some point end up with a working reimplementation that you can compare to the original.

30. Find a Creative Solution

How? Declare your bug a feature or go ahead and delete the whole functionality it affects. You can also just let the bug be and ignore it.

Why? You're left with these options if you can't fix the bug in a reasonable time frame. Whether it's realistic to use them very much depends on the bug's effects and the importance of the affected functionality.

When you realize you can't fix the bug you're working on at all or in time, you need to get creative. Of course, you should first try all other potential ways to fix the bug by employing the other strategies in this book. However, when you're out of ideas, you must start thinking outside the box.

Your users typically don't know your program's complete feature set in advance. They'll expect two types of features: key components that the program needs to fulfill its purpose and features you've already advertised. While many other features make a program better, you can change, remove, or delay them without too much of an impact.

Critical situations may also require you to change key features or those already in the users' hands. Be very cautious in such circumstances, openly communicate your actions, and provide an update as soon as possible. Fortunately, these days you can almost always update programs after release.

Don't mistake the following solutions for simply doing nothing. You still have to invest some time to manipulate the bug to your liking. These fixes are usually temporary, but if they work well and your users like them, they may last a while. They also only work in some instances, and you may run into unexpected effects. Be careful and watch out for changes in an unrelated program area that masking a bug might cause.

Declare It a Feature

While all bugs are unintentional, sometimes one can actually improve your program's overall experience or usefulness. For example, the bug may allow users to take a shortcut from one section of the program to another, display an interesting visual effect, or make the software accept and correctly process inputs it wasn't designed for.

When you realize a bug makes a great addition to your program, you should declare it an official feature and act like you planned it all along. That doesn't mean you have to advertise it to your users: You can mark it internally, so nobody wastes their time trying to find a solution and let it fly under the radar otherwise. Users will ideally just ignore any bug that doesn't interrupt their experience.

Implement It Properly

If you make a bug part of your feature set, you may have a problem when its behavior shifts. Bugs often come from dependencies you don't intend or even know about. As soon as one changes, the bug might change too, and you won't know how to undo that shift in your new feature.

For example, a bad pointer could cause a bug if it's off by a few bytes. A future developer could shift the layout of the data that pointer indicates by declaring an element on a different line, which then makes reading a number from the pointer's address yield vastly different results. If you use that number for further calculations or accessing other memory regions, a tiny change can have a catastrophic effect.

If a bug creates essential features, you need to intentionally reimplement its behavior, which requires you to clean the slate by understanding and fixing the problem. Unreliability in smaller or less central features is less of an issue, though. Your users might not even notice or care enough to be upset if a minor feature changes because either they don't use it or it still works well enough to fulfill their needs.

Determine Its Importance

To make an informed decision, you have to investigate how reliable the bug's behavior is and how important the feature might become for your users. You can get an indication of the reliability from the bug's reproducibility, which should be part of the original report, or work with your QA team to gather more complete data. Ask the team to repeatedly test the bug on all your target platforms and determine the reproduction rate specifically.

To find out what the feature's importance might be, you can talk to the people who know your users best. Your customer support and marketing colleagues probably have a good grasp of what excites your audience and

your QA team uses the program a lot. Your own teammates (or even you) may be *dogfooding*—using the program themselves before it’s released to the public, so they may know how a newfound feature will fit in.

Remove the Feature

Sometimes, a bug is so nasty and difficult to fix that it’s impossible either to live with it or to fix it with the given resources in the given time frame. It might crash a critical system, block access to a vital feature, or cause problems that lead to data loss. In short, it keeps your audience from using the program for its intended purpose or has a lasting negative impact on the program’s reputation or the users’ property.

Given an unlimited amount of resources and an unlimited amount of time, you could fix any bug. In the worst case, you would just repeatedly rewrite the whole program from scratch until it works properly. Reality limits the number of available developers (usually just you) and imposes hard deadlines (usually as soon as possible).

Bug fixing often comes at the end of the development cycle, which means you have to get it done as quickly as possible. Additionally, you want to address critical bugs fast because of their impact on the development team, users, or both. All of this means you generally don’t have much time to find a solution.

Sometimes, the only available option is to remove the feature or system containing the bug. Once it’s gone, you can release a new version to your users or let your colleagues know so they can continue their development. As removing a feature is probably only a temporary workaround, this lets you take more time to fix a bug properly.

IN PRACTICE: REMOVE OPTIONAL FEATURES

When we were working on a 3D renderer with Vulkan support, we noticed that creating pipeline state objects (PSOs) could take quite a long time. Since we had thousands of PSOs, we decided to store them in a cache on disk and reload it later, but loading sometimes made our program crash on startup. Despite spending a lot of time on debugging, we couldn’t find a solution and ultimately disabled the storage. Because it was only an optimization, we didn’t have to worry about any missing public features.

If another part of the program contains your bug's root cause, removing the feature with symptoms won't necessarily help. If another feature breaks after you take one out, you'll need to keep investigating, ideally with further experiments (see "Experiment" on [page 134](#)).

You can't really remove features that are core to your program's use case, or those which you've advertised heavily, without potentially facing false advertising accusations. Still, a temporary removal accompanied by transparent communication can work. While this doesn't speak well for the quality of software in general, users are accustomed to living with bugs and dealing with their impacts. If you're careful about both removal and communication, they may be understanding, especially if they're heavily invested in the program.

Beware of Dependencies

Watch out for any dependencies when removing a particular portion of your program, especially since you might not know about all of them. These dependencies can be technical or logical.

In technical dependencies, you've removed a function that another part of your program uses. Such an issue might show up when compilation fails or, worse, when the program crashes at runtime after calling a function it can't find. If removing one feature makes another one unstable or useless—like a text editor without the ability to save files or a game with treasure in high places but no jumping feature—you have logical dependencies.

You'll have to fix dependencies that cause crashes or affect core features, but you can live with others until you find a solution to the original problem. In some cases, you may need to remove dependencies along with the relevant feature. Be aware of potential dependencies and include them in your communication with users.

Disable, Don't Delete

Once you've decided to remove a feature, you should do it cleanly and responsibly. Don't just delete it from your codebase. While you still have a copy in your version control system in that case, it's far away from the rest of the program. You can theoretically dig it out, but that's an effort you should put in only if you really have to.

Keeping a feature around has several benefits. First, removal is often temporary, and bringing the feature back is easier and faster if you just need to reenable it rather than finding a previous commit and reversing it, especially since your code might have changed in the meantime.

Having the disabled feature in your codebase also serves as documentation. Other developers will know it existed and how it was implemented. If they want to create a comparable or related feature, they can base it on the disabled code rather than creating it from scratch. If they don't want to use the code—understandable, since it contained a bug—they can still consult it, use parts, and perhaps learn how to avoid the same bug.

Generally, you'll have three ways to disable code:

Comment it out

Commenting out the code is an obvious choice, but it also comes with problems. Some languages don't have block comments, so you must comment out each line individually. Other languages have trouble with nesting, which could cause trouble if your code already contains comments. Also, commenting out makes it difficult to spot the actual code.

The upside is that commenting out works independently of the code's hierarchy. You can use it to disable code inside a function and remove complete functions, declarations, definitions, or even just parts. You must be careful not to break the structure of the code, but commenting is a powerful tool.

Disable it at compile time

If your language supports it, disabling the code at compile time has the same characteristics as commenting it out, but with a couple advantages. You can tell it apart from comments more quickly, and you may be able to use conditions to control whether it's disabled.

In C and C++, you can do this using the `#if 0 ... #endif` preprocessor commands. Instead of the constant `0`, you can also use a `define`, which allows you to disable or enable code in various locations across files at once like so:

```
#define FEATURE_A_ENABLED 0 // Change this to 1 to enable i  
t.
```

```
#if FEATURE_A_ENABLED
--snip--
#endif
--snip--
#if FEATURE_A_ENABLED
--snip--
#endif
```

Disable it at runtime

Disabling code at runtime is a great option as well, though it has two significant downsides. First, you can only use this approach inside of functions, which means it doesn't remove type or class definitions or declarations, for example. Second, evaluating the condition at runtime means the disabled code can still affect the rest of the code.

However, the compiler still sees the disabled code, and it'll parse it so you can always be sure that it compiles. If a change elsewhere has an adverse effect on the disabled code, the build will fail. The code remains readable because your IDE applies syntax highlighting and indexes it. This approach also allows disabling code in multiple places at once if you're basing the condition on a globally shared variable.

You can avoid any performance impact from the disabled code by using a condition that is both constant and always false. This approach makes the compiler remove the code during its optimization stage because it won't ever be executed anyway. Use `if (false) {...}` to disable the code. C++'s `constexpr` can also let you put more general conditions in there, which the compiler then evaluates to permit in-optimization removal.

Always comment on any disabled code. Make sure anyone who reads it in the future knows why you disabled it and under what circumstances it may make sense to bring it back.

Enable Testing

Removing a feature entirely because it doesn't work correctly is a rather drastic measure. You may want to disable a feature only on specific hardware configurations, in certain regions, at certain times, or for certain

types of users, or disable it by default but let people enable it on demand. (You can keep the method of doing so obscure to limit access.)

Making it possible to enable a broken feature helps with debugging it. You or your QA team can test such a feature on a wide range of devices or in a live environment with distributed builds. You can also ask users to enable the feature and report back to you. All of this lets everyone else use the program without issues.

Remember to Communicate

Whether you remove a feature or turn a bug into one, always keep your colleagues and users in the loop. They have different communication needs, but you should think about them when making a decision that affects their work on the program or their use of it.

Get Consent from All Stakeholders

You should discuss changing the feature set of your program—even if it's only temporary—with all stakeholders before going through with it. A stakeholder in this case is anyone in your organization who would be impacted by the change, which could include your technical lead, the marketing department, or a director.

How much information the stakeholders will want depends on the extent of the change's impact. They likely won't be able or willing to spend much time learning about or discussing a minor change. When it comes to the program's core functionality, though, they'll want to weigh all options from delaying the release to going through with the feature set change.

Present all facts in order to get a reliable decision. Would the change affect any dependent features? What is the timeline for an eventual proper bug fix? How will the program's users potentially react? Getting buy-in from the relevant stakeholders will protect you from being the lone culprit in case any unforeseen complications arise.

Document the Decision

You must write down what you've done, why, and how. Explain the decision you've made and its impact. Mention who else participated in the process. Go into detail about why the decision makes sense and what

circumstances might call for repealing it. Also, describe its implementation in technical detail.

The documentation helps your team understand the current situation better and helps them (and you) reconstruct the decision-making process in the future. Without it, people could try fixing a bug that became a feature or cause trouble by trying to fix a disabled element.

Add documentation to the central repository and make sure the rest of the team sees it. Inform them about your decision in other ways, such as in chats and status meetings, to reach as many people as you can (see “Share What You’ve Learned” on [page 186](#)).

The code changes you make must reference this documentation. Insert a link in the comments so anyone wondering about peculiar changes can quickly read up on what you’ve done and why.

Let Your Users Know

If your solution requires users to change their interaction with your program, you need to brief them about it. You should manage their expectations, too. Tell them when you expect to bring back a working version of the feature you’ve removed. Often, they’ll just appreciate knowing that you’re working on the bug. Delaying the fix tends not to be a big problem, but it’s a good idea to set a pessimistic date and potentially surprise the users with an earlier solution.

To improve the users’ understanding of the predicament you’re in, it can help to tell them a little bit about the bug’s effects, its cause, and what you’ve already tried to do to fix it. Don’t go into too much detail, though. Vagueness protects you and your team from users trying to be helpful or accusing you of not knowing what you’re doing.

Track Your Technical Debt

Ultimately, creative, outside-the-box solutions like the ones described in this section produce technical debt: the consequences of making suboptimal changes that help you now, but that you need to address appropriately later. You must manage this debt to keep your program’s development maintainable.

Track your technical debt in a dedicated list that you share with the whole team. Add any debt you create, specific data describing it, how it

should be paid off, and where it is in the project. Remove the entry once the bug has been fixed properly. Also, add addressing the technical debt as a task in your project management tool so you won't forget it later on.

Just Let It Be

You always have to evaluate a bug against the utility of the feature it affects. Once you've spent some time investigating, you usually have a good idea of how much time and effort a solution will take and can prioritize it accordingly. Teams regularly perform bug triage leading up to a program's release to ensure they focus their efforts properly.

It's common for any kind of software to ship with many known bugs, perhaps thousands or even tens of thousands for a large, complex program. Users may never notice them because the developers fix the ones that occur frequently and have severe impacts.

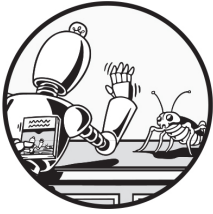
There are bugs that you see because you spend a lot of time with the program and look at it with an eye on the details. Your users may not notice more minor issues at all and, if they do, may not know that they're unintentional. However, you should add all bugs to the list of known issues you distribute with the program. This helps you avoid unnecessary bug reports.

Summary

By now the bug is fixed or at least handled. Ideally you fixed the root cause cleanly: a focused, durable change that removes the bug and everything it caused. But a perfect fix isn't always possible, and this part gave you the alternatives for when it isn't. You can ship a quick workaround and come back for the real fix later, rewrite the broken code from scratch, or find a creative way around the problem when a direct fix is out of reach.

Whatever route you took, the work isn't done until you've confirmed it. Test the fix yourself, then have someone else verify it, ideally the person who reported the bug, in the same environment where it appeared. And once it's fixed, make the bug count: share what you learned, trace how it got in, and look for the same problem elsewhere before it bites again.

CONCLUSION



You've now walked the whole workflow, from first reproducing a bug to finally fixing it, and picked up a range of strategies along the way that make you a sharper debugger and, because of it, a more effective developer.

Debugging has a reputation as the grubby part of software development, the thing you only do because you weren't good enough to avoid the bug in the first place. That's the wrong way to see it. Debugging is how you come to truly understand a program, its innermost behavior, the gap between what you think it does and what it actually does. It also brings you closer to your users. To reproduce a bug, you have to use the program the way they do and feel what they feel, which makes you appreciate exactly what the solution is worth.

Debugging also draws your team together. You collaborate on the hard problems, help each other out, and learn from a tough bug long after it is fixed—the best war stories keep people talking for years. And done well, debugging is genuinely satisfying: When you fix the highest-impact bugs first, every fix you ship is a visible win.

The rest of this conclusion looks back at the workflow, points you toward strategies beyond these pages, shows how the same thinking applies well outside of code, and leaves you with a few habits worth carrying forward.

The process of debugging relies on your imagination. Diligently follow the workflow and document everything, but when you have to choose the next strategy, the next place to investigate, or the next potential fix, you

need to be creative. You can draw from your experience, but you must also think outside the box, especially when you've hit a roadblock and feel like giving up.

What You've Learned

This book revolved around the cyclical debugging workflow: reproduce, probe, examine, and fix. Following it helps structure your work into concrete steps that you can execute iteratively, since debugging is an inherently iterative process. Each step presents several strategies without locking you into a fixed order. What you've learned from one approach may feed into the next and so on.

Your goal is to acquire knowledge about your program and its sources to produce the best possible version. While you might start the debugging process with a solid understanding of almost everything about the program (especially if you created it), the complexity of programming languages, environments, and functionalities means you'll encounter gaps in information, even if those gaps are relevant only to the immediate bug.

Reproducing the bug proves it's real and sets up everything that follows. Until you can make the bug happen on demand, you can't investigate it; you're just guessing at the cause.

Probing your program lets you rule out many areas as the bug's possible origin, narrowing your search to a few smaller sections that either contain the root cause or point you toward it.

With a relevant section in hand, you examine it closely. This is where you dig into the details of how your program actually behaves. You can loop back and forth between examining and probing until you reach the core issue.

Once you've found the root cause, you can fix it so that any bad behavior is gone for good. The fix might be a one-line change, or it might be complex and reach into surprisingly large parts of your code. Either way, make the change carefully and ship an update to your users.

Beyond This Book

Without a doubt, developers use many more strategies to fix bugs in their programs than the ones I've covered here. This book contains the tried-and-

true methods that have been most helpful throughout my career.

You likely already have, or will develop, a list of your most useful strategies. Talk about them with your team and spread the knowledge; your colleagues will appreciate it. Consider sharing your techniques more widely so that the developer community as a whole can benefit. Little educational material about debugging is available, even though it's one of the activities developers spend the most time on.

Feel free to share your strategies with me as well. I'd love to hear about them and spread the word to help more people improve their debugging skills.

Apply It Elsewhere

While debugging identifies, examines, and solves problems in computer programs, you can also apply the fundamental ideas behind the workflow and most of the strategies in other situations when you need to get information out of a system you're not familiar with. For example, consider the following uses:

Verifying changes

When you implement a change into your program, you can use debugging strategies to verify that it has the intended effect. For example, you can follow the interactions between different systems and check whether the data flows correctly or confirm that variables have the right values.

On the flip side, you can also ensure that your changes cause no unintended side effects. Comparing the runtime behavior and data from before you made the change to their equivalents afterward allows you to spot any differences.

Optimizing performance

Profiling and performance optimization are often seen as different kinds of work, but ultimately they're very similar to finding a bug's root cause. The symptom is bad runtime performance, and the *profiler* is a special tool that helps you identify where your program spends most of its time.

Using the profiler to track the problems to a specific spot in the source code corresponds to the probing step. From there, you can employ other strategies to figure out why that location is slow.

Real-world debugging

It happens to everyone: An appliance or gadget breaks, or the water or electricity stops working. All these situations call for a structured approach. First, you ensure the device doesn't actually work anymore, investigate where the problem lies, and then examine the broken part. Finally, if you have the right tools, you try to fix it.

Familiarizing yourself with a new project

When you're new to a project, you must learn its overall structure and detailed functionality. In addition to reading the source code, consulting the existing documentation, and talking to your colleagues, debugging is a great way to find out how a program behaves. That's also why bugs are commonly assigned tasks for newcomers to a project. Debugging strategies help you look at a program from different perspectives.

IN PRACTICE: DEBUG IN YOUR EVERYDAY LIFE

One time, my family came home from vacation to find no power in our house. Instead of calling an electrician immediately, I decided to take matters into my own hands and started investigating. Once I found out that everyone else in the neighborhood had functioning electricity, I knew the problem must be inside our house, so I checked the fuse box. Sure enough, the earth-leakage circuit breaker (ELCB) had tripped. I switched it back on, and it immediately tripped again. The same thing kept happening when I tried a couple more times.

Next, I switched off all other fuses. I switched the ELCB on, and it was fine. That meant one of the circuits in our house had to be tripping it. Switching each fuse back on individually eventually led me to identify that our central heating unit was the problem. That led to a call to the heating contractor, but at least I skipped an electrician.

Preventive Measures

While this book is all about what you can do after you've made a mistake, it's important to try to avoid doing so in the first place. To reduce the number of mistakes your team makes during implementation, you should

educate them as best as you can. Even if you aren't a manager or lead, you can still share useful knowledge and lead by example. Ensure your team knows the programming language and all the third-party software you use. Make them aware of common mistakes and build programs in an error-resistant way by choosing simpler designs, using defensive programming techniques, and implementing failsafe functionality.

Implement reviews as a workflow where at least one other person checks every new creation. This can be a peer review process or involve senior or leadership personnel. Both have their advantages and disadvantages.

A continuous integration system that builds your program constantly can identify potential build issues. Automated tools can also catch potential bugs before they make their way into your program. A developer can run the automation locally, or it can be part of the continuous integration.

Finally, you need a QA team that continuously tests your program to find bugs as early as possible and keep them from getting into your users' hands.

Read some of the many great resources on the topic and educate your fellow developers. For example, you can refer to *Code Complete*, 2nd edition, by Steve McConnell (Microsoft Press, 2004) or *The Pragmatic Programmer*, 20th anniversary edition, by David Thomas and Andrew Hunt (Addison-Wesley Professional, 2019). Moreover, use the strategies from this book to slowly establish a framework of instrumented code, which makes mistakes easier and faster to detect.

Parting Thoughts

That's it: a whole book about what to do when you've made a mistake while creating a program. I hope it gave you some ideas about what strategies you can try during your next bug hunt. Keep this book close by, and the next time you feel stuck, open it to a random page and start reading.

Here are four parting thoughts I'd like to leave you with:

Reproduce your bug

Reproducing the bug is the first thing you do and the last. You do it first because you can't investigate a bug you can't trigger, and last to confirm the fix actually worked. In between, you reproduce it

constantly. It's how you check whether each step is getting you anywhere. From start to finish, it's your most reliable companion.

Build a debuggable program

Developers spend significant time debugging. During some phases, debugging may occupy more of your schedule than developing new functionality. That's why you should build your program and create its sources in a debugging-friendly way.

Write readable source code and choose descriptive names. Add functionality you can use in case something goes wrong. Often, it's better to implement features straightforwardly if the requirements don't call for a more complex solution.

Keep practicing

When you're new to debugging, it can be challenging to get into a mindset that helps you find the root cause. Experience helps tremendously with identifying promising leads and likely entry points and origins.

Keep debugging and pay attention to how your team members work on issues. You'll start seeing similar bugs repeatedly and develop your intuition regarding problems.

Every bug has a cause

Computers aren't magical machines, so there must be a logical reason for your bug to occur. You just have to find it. With knowledge and time, you can fix any bug if you work systematically and consistently to avoid missing any relevant details.

You've got the book. Now go get the bugs.

INDEX

Numbers

010 Editor, [166](#)
32-bit architecture, [124–125](#)
3D meshes, [20](#), [23](#), [126](#), [137](#)
64-bit architecture, [112](#), [124–125](#)

A

abstraction layers, [78](#), [116](#), [130](#), [138](#)
 file access, [24](#)
 hardware, [130–132](#)
 randomness sources behind, [72](#), [74](#)
accessibility, [5](#)
access violations, [60](#), [91](#), [94](#)
ack search tool, [86](#)
Activity Monitor, [173](#)
AddressSanitizer, [169](#)
AFL++, [166](#)
ag search tool, [86](#)
AI (artificial intelligence), [31](#), [71](#)
algorithms, [28](#), [116](#), [138](#), [207](#), [208](#)
altWinDirStat, [172](#)
amalgamated builds, [20](#)
AMD Radeon GPU Profiler, [170](#)
APIs, [28](#)
 asynchronous IO, [28](#)
 foundational, [41](#)
 graphics, [130](#), [131](#), [170](#)
 implementation patterns, [42](#)
AppData directory, [89](#)
AR (augmented reality), [156](#)
archives, keeping, [145](#)
artificial intelligence (AI), [31](#), [71](#)
ASCII, [5](#), [40](#), [89](#)
assertions, [63–65](#), [67](#), [92](#). *See also* instrumenting programs
 adding, [63–64](#)
 commenting out, [204](#)

- enabling and disabling, [69](#)
- vs. error handling, [64](#)
- information from, [64–65](#)
- as programmatic breakpoints, [61](#), [104](#)
- as safeguards, [189](#)
- static, [65](#)
- triggering crash dumps, [92](#)
- verifying data with, [127](#)
- `assert()` macro, [63](#)
- assets, [20](#), [23](#), [122](#)
- assignment request, [xxi](#)
- attrition, preparing for, [198–199](#)
- augmented reality (AR), [156](#)
- automated tests, [190](#), [207](#), [211–212](#)
- automating the reproduction, [29–35](#)
 - hardcoding steps, [30–31](#)
 - immediate reproduction, [33–34](#)
 - recording inputs, [31–32](#)
 - scaling, [30](#)
 - with scripts, [31](#)
 - specific implementations, targeting, [32–33](#)
 - as test cases, [190](#)
 - when it isn't possible, [34–35](#)
 - whether to automate, [29–30](#)

B

- backporting, [52](#), [57–58](#)
- Backtrace, [97](#)
- bad input, [66](#)
- bad pointers, [7](#), [46](#), [213](#)
- Bazel, [20](#)
- Beyond Compare, [165](#)
- binary builds, testing with, [55](#)
- binary data, [107](#), [166](#)
- binary files, [87](#), [89–90](#), [165–166](#), [185](#), [202–203](#)
- binary search, [54](#), [56](#)
- Bing, [49](#)
- bit flips, [79](#)
- “black box” software, [43](#), [173](#)
- Black Duck, [168](#)
- blame function, [193](#)
- Boost.ASIO, [78](#)
- Boost.Log, [162](#)
- boxing and unboxing (C#), [121](#)
- branches, [8](#), [52](#), [58–59](#), [190](#)
- Breakpad, [96](#)
- breakpoints, [39](#), [59](#), [100](#), [103](#), [107–108](#)

- data, [112](#), [126](#)
- placement strategy, [61](#), [104–105](#)
- removing and disabling, [61](#), [108](#)
- tracking data changes with, [126](#)
- triggering, [64–65](#)
- on Windows, [65](#)
- break statement, [60](#)
- browser developer tools, [169](#)
- bug fixes
 - collaborating on, [185](#)
 - committing one at a time, [181–182](#)
 - criteria for, [180–181](#)
 - documenting, [143](#), [186–187](#)
 - early, [196–201](#)
 - quick, [201–206](#)
 - revertible, [181](#)
 - root cause, addressing, [181–183](#)
 - side effects, testing for, [185–186](#)
 - verifying, [5](#), [182](#)
- bug reports, [11–15](#), [53](#), [83–84](#)
 - example, [13–14](#)
 - contents of, [12–13](#), [16](#)
- bugs
 - critical, [197](#), [214](#)
 - definition of, [xviii](#)
 - reasons to fix, [xviii–xix](#)
 - taking responsibility for, [xx–xxi](#), [200–201](#)
- bug trackers, [xxi](#), [5](#), [14](#), [49](#), [52](#)
- bug triage, [84](#), [219](#)
- build configurations, [100–104](#)
- build times, reducing, [19–23](#)
- bulk changes, [195](#)
- bus factor, [199](#)

C

- C, [7](#), [19](#)
 - assertions in, [63](#), [65](#)
 - build times, [19–20](#)
 - crash dumps, [93](#)
 - exception handling, [60](#)
 - hot reloading, [25](#)
 - network-related functions, [78](#)
 - PRNG-related functions, [76](#)
 - string formatting, [161](#)
 - time-related functions, [73](#)
 - unsafe string functions, [136](#)
- C++, [7](#), [19](#)

- assertions in, [63](#), [65](#)
- `auto` keyword, [116](#)
- build times, [19–22](#)
- compilers, [103](#)
- configuration options, [87](#)
- exception handling, [60](#)
- features to look out for, [118–120](#)
- hot reloading, [25](#)
- logging libraries, [162](#)
- macros, [120](#)
- modules (C++20), [22](#)
- network functions, [78](#)
- PRNG-related functions, [76](#)
- strings, [119](#)
- templates, [22](#)
- threading-related functions, [75](#)
- time-related functions, [65](#)
- undefined behavior, [103](#)
- C#, [19](#)
 - assertions in, [63](#)
 - boxing and unboxing, [121](#)
 - build times, [19–20](#)
 - exception handling, [60](#)
 - hot reloading, [25](#)
 - features to look out for, [120–121](#)
 - logging libraries, [162](#)
 - network-related functions, [78](#)
 - pitfalls, [120–121](#)
 - PRNG-related functions, [76](#)
 - threading-related functions, [75](#)
 - time-related functions, [73](#)
- call sites, searching for, [195](#)
- call stacks, [60–61](#), [92](#), [95](#), [105–107](#)
- Ccache, [20](#)
- changelogs, [56](#), [130](#)
- changes, implementing, [183–185](#)
- character encodings, [89](#), [124](#), [127](#)
- chat services, [49](#), [82](#), [145–146](#)
 - as debugging logs, [145–146](#)
- cheat codes, [151](#)
- Clang, [21](#), [53](#), [101–102](#)
 - disabling optimizations in, [53](#), [204](#)
 - sanitizers, [168–169](#)
 - warnings, [168](#)
- clean room implementation, [208](#)
- clean shutdowns, [7](#)
- CMake, [20](#)
- code comments, [44](#), [187](#)

- Code Complete* (McConnell), [225](#)
- code modifications, [134–135](#)
- CodePen, [167](#)
- code playgrounds, [167](#)
- code sanitizers, [168–169](#)
- coding standards, [191](#)
- command line arguments, [30](#), [46](#), [110](#), [150](#), [173](#)
- comma-separated values (CSV) files, [143](#)
- comments
 - as clue to difficult systems, [44](#)
 - commenting out code, [138](#), [155–156](#), [202](#), [204](#), [216](#)
 - on disabled code, [217](#)
 - documenting fixes in, [143](#), [187](#)
 - readable code and, [116](#)
 - searchable markers in, [22](#), [90–91](#), [140](#), [177](#), [206](#)
 - searching for similar, [194](#)
- commit messages, [54–56](#), [186–187](#), [193–194](#), [206](#)
- commits, [54–56](#), [188](#), [193–194](#)
 - identifying the offending change, [56](#)
- Compiler Explorer, [167](#)
- compile times, avoiding long, [22](#)
- compression, [24](#), [28](#), [122](#)
 - algorithms, [28](#)
 - of symbol files, [95](#)
 - texture, [23](#)
- concurrency, [74](#), [107](#)
- conditional breakpoints, [104–105](#)
- conditional statements, [61](#)
- configuration files, [20](#), [30](#), [89](#), [125](#)
- Confluence, [145](#)
- connection speed, fixing, [24](#)
- `continue` statement, [60](#)
- continuous integration system, [55](#), [168](#), [224](#)
- control flow, [59–60](#), [67](#), [102](#)
 - statements, [60](#)
- Control Flow Integrity, [169](#)
- cosmic radiation, [79](#)
- Coverity, [168](#)
- Cppcheck, [168](#)
- CPUs, [28](#), [95](#), [96](#), [104](#), [154](#)
 - architectures, [110](#)
 - core, [23](#), [71](#), [74](#)
- crash dumps, [13](#), [30](#), [38](#), [91–97](#)
 - choosing the right type, [95](#)
 - gathering data from, [91–92](#), [95](#)
 - implementing, [92–94](#)
 - with C, [93](#)
 - out-of-process handler, [94](#)

- triggering, [91–92](#)
- Crashpad, [96](#)
- creative solutions, finding, [212–219](#)
- curl, [38](#), [78](#)
- custom-built debuggers, [62](#)
- custom tools, [146–152](#)
 - implementing, [147](#)
 - for libraries, [150](#)
 - recommended, [149](#), [152](#)
 - release considerations, [152](#)
- curl, [38–39](#), [78](#)

D

- dangling pointers, [119](#)
- data, [122](#)
 - finding bad data, [126–127](#)
 - good vs. bad, [127](#)
 - original input, [123](#)
 - permanent vs. transient, [125](#)
 - symptoms of bad data, [123–126](#)
 - transformation of, [122](#)
- DataFlowSanitizer, [169](#)
- data structures, [130](#)
- DB Fiddle, [167](#)
- deadlocks, [43](#), [130](#)
- dead zones, [200](#)
- Dear ImGui, [147](#)
- Debug.Assert() method, [63](#)
- debug builds, [22](#), [68](#), [100](#), [103–104](#)
- debuggers
 - attaching, [59](#)
 - memory view, [68](#), [107](#), [125](#)
 - when not to use, [62](#), [159–160](#)
- debugging
 - advanced, [111–113](#)
 - inevitability of, [xix](#)
 - logs
 - archiving, [146](#)
 - chats as, [145–146](#)
 - documentation tools for, [144–145](#)
 - keeping, [140–146](#)
 - summarizing findings, [143](#)
 - tools, [144–145](#)
 - tracking progress in, [141–142](#)
 - rewards of, [221](#)
 - starting the process, [104–110](#)
 - workflow, [1–4](#)

- examine step, [3–4](#)
- fix step, [4–5](#)
- iterative nature of, [2](#), [99](#), [206](#), [222](#)
- probe step, [3](#)
- reproduce step, [2](#)
- DebugView, [172](#)
- decoupling, [45](#)
- delta times, [71–73](#)
- denormalized numbers, [124](#)
- dependencies, [xix](#), [8](#), [57](#), [172](#), [215](#)
 - chains, [45](#)
 - data, [4](#)
 - eliminating, [24](#)
 - tracking, [23](#), [25](#)
- Dependencies tool, [172](#)
- Dependency Walker, [172](#)
- dependent behavior, avoiding, [199–200](#)
- deployment, [8](#), [23–24](#)
- deserialization, [28](#)
 - zero-copy, [28](#)
- determinism, [xxiii](#), [70–72](#)
- diff tool, [127](#), [165](#)
- Direct3D, [41](#), [131](#)
- DirectStorage, [28](#)
- dirty room implementation, [209–210](#)
- disabling code, [155–156](#), [215–217](#)
 - commenting out, [216](#)
 - at compile time, [190](#), [216](#)
 - at runtime, [216–217](#)
- disassemblers, [165–166](#)
- Discord, [49](#)
- distcc, [20](#)
- distractions, avoiding, [143–144](#)
- distributed caching, [20](#)
- distributed compilation, [20](#)
- division by zero, [60](#), [91](#), [120](#), [124](#)
- D language, [65](#)
- dmalloc, [171](#)
- dmesg, [172](#)
- documentation
 - of bug fixes, [143](#), [186–187](#)
 - as clue to difficult systems, [44](#)
 - of debugging tools, [148](#), [152](#)
 - of decisions, [218](#)
 - of known issues, [133](#), [141](#), [219](#)
 - searching, [52](#), [89](#), [133](#)
 - of specifications, for rewrites, [208–209](#)
 - tools for, [144–145](#)

- of workarounds, [206](#)
- dogfooding, [214](#)
- dot directory, [89](#)
- drivers, [130–132](#)
- dry runs, performing, [149](#)
- DTrace, [172](#)
- DuckDuckGo, [49](#)
- duck test, [128](#)
- dynamic link libraries, [25](#), [94](#), [172](#)

E

- Electron, [41](#), [83](#)
- Elements of Programming Style, The* (Kernighan and Plauger), [xx](#)
- EMacs, [167](#)
- entry points, [3](#), [37](#), [97](#)
- enums, [67](#)
- environment changes, [205](#)
- error codes, [66–67](#)
- error-correcting codes (ECCs), [79](#)
- error handling, [64](#), [65–67](#), [69](#), [191](#)
- error messages, searching for, [50](#)
- error signaling, [67](#)
- error value, [66](#)
- Everything (search tool), [87](#), [166](#)
- exceptions, [60](#), [67](#), [118](#)
 - as control-flow statements, [60](#)
 - breaking on, in debuggers, [60](#)
 - hardware, [91–92](#)
 - working around, [204](#)
- execution pointers, [103](#)
- exit codes, [66](#)
- expectations, managing, [xxii](#)
- explicit type conversions, [118](#)
- external tools, [148–149](#). *See also* [custom tools](#)
- extern template, [22](#)

F

- FASTBuild, [20](#)
- fast-math option, [71](#)
- features disguised as bugs, [184](#)
- feedback loop, [19](#)
- Feynman technique, [173](#)
- Fiddler, [170](#)
- file access functions, [46](#)
- FileLocator Pro, [86](#)
- file locks, [8](#), [43](#), [172](#)
- File Locksmith, [172](#)

- filepaths, [33](#), [40](#), [87](#)
- file search tools, [87](#), [166](#)
- Firebase Crashlytics, [97](#)
- fixed time steps, [71–72](#)
- floating-point inaccuracies, [7–9](#)
- floating-point numbers, [7](#), [32](#)
 - delta times, [71–73](#)
 - determinism and, [71](#)
 - inaccuracies, [7–8](#), [71](#), [184–185](#)
 - special values (IEEE 754), [124](#)
- foreign language fonts, [155](#)
- forks, [58](#)
- format linters, [166](#)
- format validators, [166](#)
- forums, [49](#), [82](#)
- forward declaration, [21](#)
- frameworks, widely used, [41](#)
- frequently asked questions list, [82](#)
- full dumps, [95](#)
- full-text search, [86](#), [145](#)
- function calls, [4](#), [33](#), [112](#)
- function parameter, [106](#), [193](#)
- future-proofing, [181](#)
- fuzzers, [166](#)
- fuzzing, definition of, [166](#)
- fuzzy search, [87](#)
- `fzf`, [87](#)

G

- games
 - AI-driven reproduction in, [31](#)
 - assets, [20](#), [23–25](#)
 - debugging commands for, [151–152](#)
 - determinism in, [71](#)
 - saved games as shortcuts, [34](#)
- garbage collection (C#), [121](#)
- GCC, [21](#), [168](#), [169](#)
- GDB, [113](#)
- Ghidra, [166](#)
- `git bisect` command, [54](#)
- Gitflow, [58](#)
- GitHub, [51](#), [57](#), [83](#)
- `git revert` command, [202](#)
- global variables, [22](#), [107](#)
- Godbolt, [167](#)
- god mode, [151](#)
- Go language, [52](#), [113](#)

- `goto` statement, [60](#)
- graphics APIs, [41](#), [130–131](#), [170](#)
 - debugging tools for, [170–171](#)
 - validation layers, [170–171](#)
- graphics cards, [85](#), [96](#), [170](#)
- graphics processing units (GPUs), [170](#)
- graphics tools, [170–171](#)
- `grep`, [86](#)
- guard values, [67](#)
- GUI, [18](#), [20](#)
- gyroscopes, [32](#), [205](#)

H

- Hacker News, [49](#)
- `handle_exception()` function, [93](#)
- hardware abstraction layers, [130](#)
- hardware issues, [37](#), [79](#), [132–133](#), [205](#)
- header files, [20](#), [22](#)
- help, asking for, [79–85](#)
- hexadecimal, [68](#)
- hex editors, [90](#), [166](#), [203](#)
- Honggfuzz, [166](#)
- hot reloading, [25–26](#)
- HTTP requests, [24](#), [115](#), [138](#)
- Hunt, Andrew, [225](#)
- HxD, [165](#)
- hybrid tools, [149](#). *See also* [custom tools](#)
- hypotheses, [134](#)
 - confirming or rejecting, [137](#)
 - examples, [138–139](#)
 - formulating, [135](#)
 - testing, [135](#)

I

- IDA Pro, [166](#)
- IDE (integrated development environment), [38](#), [61](#), [86](#), [102](#), [103](#)
- IEEE standard, [71](#), [124](#)
- `if` statement, [59](#), [61](#), [64](#)
- immediate mode GUI (IMGUI), [147](#)
- immediate reproduction, [33](#)
- implicit type conversions, [118](#)
- improving your code. *See* [readable code](#)
- `#include` directive, [20](#)
- include-what-you-use, [21](#)
- Incredibuild, [20](#)
- indirection, [xxiii](#), [3](#), [8](#)
- infinite loops, [43](#), [141](#)

infinite recursion, [106](#)
InfluxDB, [41](#)
inlining, [102](#)
input events, [31–32](#), [43](#)
 recording, [31–32](#)
integrated development environment (IDE), [38](#), [61](#), [86](#), [102](#), [103](#)
integrated tools, [147–148](#). *See also* [custom tools](#)
interdependencies, [18](#), [188](#)
internet searches, [47–52](#)
inter-process communication, [77](#)
intranets, [52](#)
invalid memory access, [119](#)
IP addresses, [163](#)
irrelevant systems, [45–47](#)
ISO standard, [xviii](#)
iteration times, [16](#), [33](#), [135](#), [155](#), [207](#), [211](#)

J

Java, [19](#), [113](#), [115](#), [132](#), [168](#)
JavaScript, [20](#), [115](#), [167](#), [169](#)
JSFiddle, [167](#)
JSON, [124](#)
jumbo builds, [20](#)

K

KDiff3, [165](#)
Kernighan, Brian W., [xx](#)
known issues lists, [133](#), [141](#), [219](#)

L

launchers, [110](#)
LeakSanitizer, [169](#)
libFuzzer, [166](#)
LinkedIn, [85](#)
link-time optimization (LTO), [22](#)
Linux, [73](#), [113](#)
Live++, [25](#)
LLDB, [113](#)
local variables, [106](#)
log files, [30](#)
logging libraries, [162–163](#)
logs, tools for, [172](#)
log statements, [68](#), [158](#)
Loguru, [162](#)
Lua, [20](#), [115](#), [160](#)

M

- magic numbers, [67–68](#), [116](#)
- magic values, [67–68](#)
- Maguire, Steve, [67](#)
- `main()` function, [33](#), [46](#), [61](#), [93](#), [114](#), [139](#)
- `make` tool, [110](#)
- markdown-formatted text, [144](#)
- masking bug symptoms, [180](#), [203](#), [205](#), [213](#)
- Mastodon, [49](#)
- Mathematica, [117](#)
- matrix multiplication, [135](#)
- McConnell, Steve, [225](#)
- MediaWiki, [145](#)
- member variables, [106–107](#)
- Memoirs of a Computer Pioneer* (Wilkes), [xix](#)
- memory
 - address, [7](#), [112](#)
 - allocation, [74](#)
 - blocks, [107](#)
 - corruption, [46](#), [68–69](#), [103](#), [107](#)
 - leaks, [119](#), [169](#), [171–172](#)
 - mapping, [28](#)
 - tools, [171–172](#)
 - uninitialized, [68](#), [73–74](#), [103](#), [169](#)
 - view, in debuggers, [68](#), [107](#), [125](#)
- memory_profiler, [171](#)
- MemorySanitizer, [169](#)
- menus, removing at startup, [27](#)
- Metal (graphics API), [52](#), [131](#), [170](#)
- Microsoft Docs, [49](#)
- Microsoft Teams, [145](#)
- middleware, [48](#), [57](#), [148](#), [155](#)
- mini dumps, [95](#)
- `MiniDumpWriteDump()` function, [95](#)
- minimal repos, building, [153](#)
- `mmap()` function, [28](#)
- mocking, [115](#)
- model-view-controller (MVC) pattern, [42](#)
- MongoDB, [41](#)
- Morrowind*, [204](#)
- Mozilla Developer Network, [49](#)
- MSVC, [21](#), [168](#)
- multitasking, [74](#)
- multithreading, [28](#), [33](#), [41](#), [44](#), [69](#), [71](#), [75](#), [107](#), [204](#)
- mutexes, [43](#), [75](#), [204](#)
- MySQL, [41](#)

N

- NaN (Not a Number), [124](#)
- Natvis file, [111](#)
- negative infinity, [124](#)
- networking
 - complexity, [18–19](#)
 - functions, [78](#)
 - tools, [169–170](#)
- NLog, [162](#)
- nondisclosure agreements, [83](#), [174](#)
- nontext files, [89](#)
- Not a Number (NaN), [124](#)
- Notepad++, [167](#)
- Notion, [145](#)
- Nsight Aftermath, [95](#)
- Nsight Graphics, [153](#), [170](#)
- null pointers, [7](#), [64](#), [91](#), [109](#), [114](#), [124](#)
- Nvidia, [85](#), [95](#), [153–154](#), [170](#)

O

- objdump, [166](#)
- OBS Studio, [166](#)
- off-by-one error, [7](#)
- off-the-shelf tools, [164–173](#)
 - availability of, [165](#)
 - code playgrounds, [167](#)
 - code sanitizers, [168–169](#)
 - diff and merge tools, [165](#)
 - disassemblers, [165–166](#)
 - general-purpose, [165–167](#)
 - graphics tools, [170–171](#)
 - memory tools, [171–172](#)
 - networking tools, [169–170](#)
 - process tools, [173](#)
 - static analyzers, [167–168](#)
 - system tools, [172–173](#)
 - when to reach for, [165](#)
- Oodle, [28](#)
- OpenGL, [85](#), [131](#)
- open source software, [20](#), [57–58](#), [77](#), [96](#), [130](#), [165](#)
- operating systems
 - as environment for bugs, [132–133](#)
 - input recording at OS level, [32](#)
 - scheduler, [71](#), [74](#)
 - tracing tools, [172](#)
- operator overloading, [118](#)
- optimization, effects of, [100–103](#)
- optimization levels, [22](#), [102](#)

Overlapped IO, [28](#)
overlaying assets, [24](#)
ownership, taking, [xx–xxi](#), [200–201](#)

P

P4Merge, [165](#)
pair programming, [208](#)
passwords, [157](#), [163](#)
.pdb format, [94](#)
Perl, [113](#)
persistent data, [199](#)
personal data, [xix](#), [157](#)
PHP, [20](#)
PIMPL (pointer-to-implementation) idiom, [21](#)
PIX, [170](#)
plaintext files, [144](#)
Plauger, P.J., [xx](#)
pointers, [7](#), [74](#), [107](#), [119](#), [124–125](#)
pointer-to-implementation (PIMPL) idiom, [21](#)
polymorphism, [118](#)
POSIX, [75](#), [78](#)
positive infinity, [124](#)
PostgreSQL, [41](#)
Postman, [170](#)
postmortem meetings, [189](#)
Pragmatic Programmer, The (Thomas and Hunt), [225](#)
precompiled header files, [20](#)
pre-conditions and post-conditions, [41](#), [44](#), [63](#), [192–193](#)
preprocessor commands, [216](#)
presentations, preparing, [177](#)
printf, [158–164](#)
 cleaning up, [163–164](#)
 implementing, [160–162](#)
 logging, [162–163](#)
 pitfalls, [159](#)
 statements, [158](#)
 when to use, [159](#)
privacy, [xix](#), [163](#)
PRNGs, [70](#), [76](#), [77](#)
Process Explorer, [110](#), [172](#), [173](#)
Process Hacker, [173](#)
Process Monitor, [172](#)
process tools, [173](#)
programming languages, [117–122](#)
project history, checking, [52](#)
project management tools, [xxi](#), [52](#), [89](#), [143](#), [187](#), [194](#), [200](#), [219](#)
ps command, [110](#), [173](#)

pseudo-random number generators (PRNGs), [70](#), [76](#), [77](#)

Pthread, [75](#)

PVRTexTool, [171](#)

PVS-Studio, [168](#)

Pylint, [168](#)

Python

- build times, [20](#)

- features to look out for, [121–122](#)

- hot reloading, [25](#)

- network-related functions, [78](#)

- PRNG-related functions, [76](#)

- threading-related functions, [75](#)

- time-related functions, [73](#)

- versions, [122](#)

Q

quality assurance (QA) teams, [xxi](#), [5–6](#), [12](#), [185–186](#), [195–196](#), [214](#), [224](#)

Qt, [83](#)

Questioning everything, [127–134](#)

quick fixes, deploying, [201–206](#)

Quill, [162](#)

R

race conditions, [46](#), [62](#), [169](#)

`RaiseException()` function, [91](#)

RAM, [79](#)

`rand()` function, [47](#), [76](#)

randomness, [69–79](#)

random number generators, [75](#). *See also* [pseudo-random number generators \(PRNGs\)](#)

raw data, inspecting, [171](#)

React, [41](#)

readable code, principles of, [115–117](#)

reading code, [113–122](#)

read-only files, [8](#), [43](#)

Reddit, [49](#), [83](#)

refactoring source code, [183](#)

Regex101, [88](#)

RegExr, [88](#)

regressions, preventing, [158](#), [190](#)

regular expressions, [87–88](#), [90](#)

remote debugging, setting up for, [176](#)

removing features, [214–217](#)

RenderDoc, [152](#), [170](#)

reports. *See* [bug reports](#)

reproducing bugs, [xix](#), [2](#), [11](#), [15–19](#), [214](#)

- emulating the reporter, [16–17](#)

- frequency of, [xix](#), [12](#), [69](#)

- networking complexity and, [18–19](#)
- questions to ask, [14](#)
- re-creating the user experience, [15–16](#)
- variations, introducing, [17–18](#)
- verifying fixes by, [5](#), [182](#), [185](#)
- steps to, [2](#), [12](#), [17–18](#), [27](#), [30–31](#)
- restarting programs, [204](#)
- return statement, [60](#), [103](#)
- return values, [66](#)
- rewriting functionality, [206–212](#)
- ripgrep, [86](#)
- roadmaps, public, [50](#)
- root causes, [xxi](#), [xxiii](#), [3](#), [4](#)
 - addressing, [181](#)
 - common, [6–9](#)
 - determining, [187–189](#)
 - removing, [182–183](#)
- Roslyn Analyzers, [168](#)
- rubberducking, [175](#)
- runtime filtering, [164](#)
- runtime switches, [186](#), [211](#), [216–217](#)
- Run to Cursor feature, [108](#)
- Rust, [19](#), [51–52](#), [65](#), [113](#), [167](#)
- Rust Playground, [167](#)

S

- saved state, [13](#), [33–34](#)
- screen recorders, [166](#)
- screenshots, [13](#), [16](#), [142](#), [166](#)
- ScreenToGif, [166](#)
- scripting languages, [25](#), [31](#), [60](#)
- searchable patterns in code, [90](#)
- search engines, [47–52](#)
 - advanced features of, [50–51](#)
 - operators, [50–51](#)
 - source code, searching for, [51–52](#)
 - trying different, [49–50](#)
- searching code, tools for, [86](#)
- searching files, [86–91](#)
 - search terms, [87–88](#)
 - search tools, [86](#)
 - configuring, [87](#)
 - Everything, [87](#)
 - FileLocator Pro, [86](#)
 - fzf, [87](#)
 - grep, [86](#)
 - ripgrep, [86](#)

- seed, [76](#)
- semaphores, [75](#)
- sentinels, [67](#)
- Sentry, [97](#)
- serialization, [29](#), [33–34](#)
- shaders, [23](#)
- Shadertoy, [167](#)
- shared network directory storage, [95](#)
- sharing what you’ve learned, [186–191](#)
- shortcuts, [38–47](#)
- side effects, testing for, [185–186](#)
- SIMD instructions, [102](#)
- simultaneous events, [32](#)
- singletons, [107](#)
- Slack, [49](#), [145](#)
- social media, [48](#), [49](#), [83](#), [84](#)
- software development, [43](#)
- source code licenses, [57](#)
- spdlog, [162](#)
- special characters in code, [51](#)
- splash screens, [26](#)
- Spotlight, [166](#)
- sprints, [58](#)
- SQL databases, [167](#)
- SQL Fiddle, [167](#)
- SSDs, [27](#), [87](#), [198](#)
- Stack Exchange, [83](#)
- Stack Overflow, [49](#), [83](#)
- stack overflows, [106](#)
- stakeholder consent, [217](#)
- startup times, reducing, [26–29](#)
- static analyzers, [167–168](#)
- static variables, [106](#)
- step into commands, [61](#)
- step out commands, [61](#)
- stepping through code, [107–108](#)
- stopgap solution, [183](#)
- strace, [172](#)
- `strcmp()` function, [119](#)
- Street Fighter II*, [200](#)
- string concatenation, [51](#), [90](#)
- stripped builds, [94](#)
- structlog, [162](#)
- structured exception handling (SEH) extension, [60](#)
- stubs, [212](#)
- Sublime Text, [167](#)
- suspicious systems, [43–45](#)
- sustainable resolution, factors affecting, [180](#)

- symbol files, [94–95](#), [111](#)
- symptoms, [12](#), [108](#)
 - of bad data, [123–126](#)
 - masking with workarounds, [203–205](#)
 - tracing to root causes, [108](#)
- System Informer, [110](#)
- System Monitor, [173](#)
- system tools, [172–173](#)

T

- talking through bugs, [173–178](#)
- Task Manager, [173](#)
- task schedulers, [44](#), [46](#), [75](#)
- teams, working as, [5–6](#)
- teardown functions, [77](#)
- technical debt, [39](#), [188](#), [201](#), [218](#), [219](#)
- templates (C++), [22](#)
- temporary changes, reverting, [53](#)
- temporary objects (C++), [119](#)
- temporary solution. *See* [quick fixes](#), [deploying](#)
- testing, [210](#), [217](#)
- test patterns, [67–68](#)
- text editors, [86](#), [167](#)
- texture compression, [23](#)
- third-party software
 - crash reporting tools, [96–97](#)
 - debugging, [129](#), [153](#)
 - fixes in newer versions of, [56–58](#)
 - integrating fixes from, [57–58](#)
 - source code licenses for, [57](#)
 - support from, [82–84](#)
 - symbol files for, [95](#)
- Thomas, David, [225](#)
- threading primitives, [46](#)
- threading-related functions, [75](#)
- ThreadSanitizer, [169](#)
- time-related functions, [73](#)
- time step, [71](#)
- time zones, [7](#), [132](#)
- transformations, [126](#)
- transient data, [125–126](#), [199](#)
- try-catch block, [204](#)

U

- UDP, [170](#)
- UndefinedBehaviorSanitizer, [169](#)
- underlying dependencies, [192–193](#)

Unified Modeling Language (UML), [142](#)
uninitialized memory, [68](#)
unity builds, [20](#)
Unreal Engine, [20](#), [58](#), [83](#)
update rates, [71](#)
urgency, level of, [5](#)
user experience (UX), [9](#)
 determinism and, [70](#)
 issues as root causes, [9](#)
 re-creating, [15–16](#)
UTF-8, [5](#), [89](#), [127](#)
UTF-16 and UTF-32, [127](#)

V

Valgrind, [171](#)
validation layers, [170](#)
variables, naming, [90](#)
`/var/log/` directory, [172](#)
verbose mode, [149](#)
version control history, [52–59](#), [193–194](#)
vim, [167](#)
virtual machines, [19–20](#), [25](#), [113](#), [132](#)
 acting like, when reading code, [113–115](#)
Visual Studio, [20](#), [25](#), [60](#), [68](#), [86](#), [111–112](#)
Visual Studio Code, [167](#)
`void` pointers, [22](#)
VsChromium, [86](#)
VTune, [152](#), [172](#)
Vulkan, [41](#), [131](#), [170](#), [215](#)

W

web development tools, [167](#), [169](#)
wikis, [52](#), [145](#)
Wilkes, Maurice, [xix](#)
WinDirStat, [172](#)
Windows Error Reporting (WER), [96](#)
WinForms, [41](#)
WinMerge, [165](#)
Wireshark, [170](#)
WizTree, [172](#)
workarounds, [203](#), [206](#)
Writing Solid Code (Maguire), [67](#)

X

X, [49](#)
x86 instructions, [109–110](#)
XML, [23](#), [111](#), [124](#), [166](#)

Y

Y2K bug, [132](#)

YAML, [124](#)

YouTube, [49](#)

Z

zero-cost logs, [164](#)

.zip files, [89](#)

zlib, [28](#)

Zstandard, [28](#)