I do not use a debugger

I learned to program with BASIC back when I was twelve. I would write elaborate programs and run them. Invariably, they would surprise me by failing to do what I expect. I would struggle for a time, but I'd eventually give up and just accept that whatever "bugs" I had created were there to stay.

It would take me a decade to learn how to produce reliable and useful software. To this day, I still get angry with people who take it for granted that software should do what they expect without fail.

In any case, I eventually graduated to something better: Turbo Pascal. Turbo Pascal was a great programming language coupled with a fantastic programming environment that is comparable, in many ways, to modern integrated development environments (IDEs). Yet it is three decades old. It had something impressive: you could use a "debugger". What this means is that you could run through the program, line by line, watching what happened to variables. You could set breakpoints where the program would halt and give you control.

Recently, Chris Wellon wrote about Borland C++, an environment from the 1990s:

> *I made a couple of test projects, built and ran them with different options, and poked around with the debugger. The debugger is actually pretty decent, especially for the 1990s.*

At the time, I thought that programming with a debugger was the future.

Decades later, I program in various languages, C, JavaScript, Go, Java, C++, Python… and I almost never use a debugger. I use fancy tools (sanitizers, static analyzers, continuous integration), and I certainly do use tools that are called debuggers (like `gdb`), but I almost never step through my programs line-by-line watching variable values. I almost never set breakpoints. I say "almost" because there are cases where a debugger is the right tool, mostly on simple or quick-and-dirty projects, or in contexts where my brain is overwhelmed because I do not fully master the

language or the code. This being said I do not recall the last time I used a debugger as a debugger to step through the code. I have a vague recollection of doing so to debug a dirty piece of JavaScript.

I am not alone. In five minutes, I was able to find several famous programmers who took positions against debuggers or who reported barely using them.

- Linus Torvalds, the creator of Linux, does not use a debugger.
- Robert C. Martin, one of the inventors of agile programming, thinks that debuggers are a wasteful timesink.
- John Graham-Cumming hates debuggers.
- Brian W. Kernighan and Rob Pike wrote that *stepping through a program less productive than thinking harder and adding output statements and self-checking code at critical places*. Kernighan once wrote that *the most effective debugging tool is still careful thought, coupled with judiciously placed print statements*.
- The author of Python, Guido van Rossum has been quoted as saying that uses print statements for 90% of his debugging.

I should make it clear that I do not think that there is one objective truth regarding tools. It is true that our tools shape us, but there is a complex set of interactions between how you work, what you do, who you work with, what other tools you are using and so forth. Whatever works for you might be best.

However, the fact that Linus Torvalds, who is in charge of a critical piece of our infrastructure made of 15 million lines of code (the Linux kernel), does not use a debugger tells us something about debuggers

Anyhow, so why did I stop using debuggers?

Debuggers were conceived in an era where we worked on moderately small projects, with simple processors (no thread, no out-of-order execution), simple compilers, relatively small problems and no formal testing.

For what I do, I feel that debuggers do not scale. There is only so much time in life. You either write code, or you do something else, like running line-by-line through your code. Doing “something else” means (1) rethinking your code so that it is easier to maintain or less buggy (2) adding smarter tests so that, in the future, bugs are readily identified effortlessly. Investing your time in this manner makes your code better in a lasting manner… whereas debugging your code line-by-line fixes one tiny problem without improving your process or your future diagnostics. The larger and more complex the project gets, the less useful the debugger gets. Will your debugger scale to hundreds of processors and terabytes of data, with trillions of closely related instructions? I’d rather not take the risk.

My ultimate goal when work on a difficult project is that when problems arise, as they always do, it should require almost no effort to pinpoint and fix the problem. Relying on a debugger as your first line of defense can be a bad investment, you should always try to improve the code first.

Rob Pike (one of the authors of the Go language) once came to a similar conclusion:

> *If you dive into the bug, you tend to fix the local issue in the code, but if you think about the bug first, how the bug came to be, you often find and correct a higher-level problem in the code that will improve the design and prevent further bugs.*

I don't want to be misunderstood, however. We need to use tools, better tools… so that we can program ever more sophisticated software. However, running through the code line-by-line checking the values of your variables is no way to scale up in complexity and it encourages the wrong kind of designs.

Let me end with a quote that sums up my sentiment:

> *"Debuggers don't remove bugs. They only show them in slow motion."*

Further reading: Ben Deane, an experienced game developer who worked on Goldeneye, Medal of Honor, StarCraft, Diable, World of Warcraft, and so forth, wrote in 2018:

> *The principal problem with debugging is that it doesn't scale. (…) in order to catch bugs, we often need to be able to run with sufficiently large and representative data sets. When we're at this point, the debugger is usually a crude tool to use (…) Using a debugger doesn't scale. Types and tools and tests do.*

PUBLISHED BY

Daniel Lemire

A computer science professor at the University of Quebec (TELUQ). View all posts by Daniel Lemire →

94 thoughts on “I do not use a debugger”

M. Eric DeFazio

June 21, 2016 at 3:06 pm

I’ve always felt like I missed the boat on the value of debuggers. A coworker once showed me how “cool” it was that he had a remote debugger running on a Server stepping through code and looking at the state of things… it occurred to me that the tool was needed BECAUSE the code was a mess, (too many “smart”/hierarchical/mutable objects interacting with each other).

I do see the value of using a debugger to understand code you don’t own…but there is always the idea that you ended up using a debugger because you really don’t understand the model and state transitions the program can go through. If that is the case, time should be spent figuring out the model and state transitions in a top down way… rather than iterating on figuring out where the model and or transitions are broken (in a bottom up way) by figuring out the appropriate places to put breakpoints. (But I guess this is an art in and of itself)

Lukas Eder

July 2, 2016 at 2:24 pm

> *time should be spent figuring out the model and state transitions in a top down way*

Excellent! Just put a breakpoint in your main() method, and step into each relevant method, seeing what’s going on.

Maxence

June 21, 2016 at 5:04 pm

A debugger is a very useful tool. I think people which are not using them is people who have catch bad habits while working in environments where there was no good debugger. When you work with Visual Studio with its marvelous debugger, you can find and resolve bugs faster. Of course, you can always think harder to the problem and review your code, but when you’re tired, the debugger is a nice helper.

Daniel Lemire

June 21, 2016 at 6:12 pm

I think people which are not using them is people who have catch bad habits while working in environments where there was no good debugger.

Good debuggers are hardly new. Turbo Pascal had a fantastic debugger three decades ago, and even then it wasn't a particularly innovative feature. A debugger is a very widely available tool and it has been so for many decades. IDEs (with debuggers) are also widely available and have been for a very long time… E.g., Visual Studio, Eclipse, Intellij, KDevelop and so forth.

There are a few small things that have improved with respect to debuggers… such as backward execution, remote cross-platform debugging, pretty-printing STL data structures… but overall, debuggers have not changed very much.

IDEs have gotten quite a bit nicer and smarter… Do IDEs make you more productive? That's another debate.

Of course, you can always think harder to the problem and review your code, but when you're tired, the debugger is a nice helper.

I agree that using tools to make yourself more effective, especially when you are cognitively impaired in some way, is very important. But in most cases there are better tools than a debugger.

Paul Carroll

March 21, 2020 at 6:22 am

The corollary to that however is that *you* may find the bug quickly on *your* machine, but when you come to debugging something on a SIT, UAT or (god forbid) PROD machine/cluster/network you are doomed.

This is why, having followed a similar career path to OP, I have switched to good old trusty debug logging where the parallelism is strikingly apparent. Terse injection of variable is immeasurably useful and rather akin to watch variables… not only that, but they translate to others that may have to maintain/debug your code, oh and they jog you memory as to how the code can/needs to be debugged.

Each to his own of course, but I enjoy my new perspective FWIW.

Henry Skoglund

June 21, 2016 at 7:25 pm

I also shy from using the debugger, main reason for me is to avoid the overhead of switching context, i.e. one minute you're in your familiar IDE editing your program, and the next you're

in a different environment with new windows (the debugger).

It's like watching 2 movies at the same time.

It's much less burden for my tiny brain to just insert some printf()s where my code is suspect and watch that output.

Stefano Miccoli

June 21, 2016 at 8:03 pm

My two cents is that for me `assert` and `print` are by far the most useful debugging tools, in a wide range of programming paradigms.

Well designed `asserts` are helpful for both catching logic errors and documenting the code. In fact `asserts` explicitly highlight assumptions about the algorithm state that otherwise could go unnoticed.

As Daniel Lemire correctly points out, one should not focus its attention on *where* the code is wrong, but on *why* the code is wrong, and `asserts` are ideal to this purpose.

As what regards `print` statements, they are the poor man's debugger, but somehow I learned to cope with the fact that I'm a poor man, without the energy for mastering a high polish modern debugger tool…

Ben

June 22, 2016 at 1:01 pm

I also try to write as many good assertions as I can in my code. There's also the new-ish thing in reliability: property based testing (a la QuickCheck). I like to think of property based testing as somewhere in the nebulous space between conventional use of assertions and full-blown formal verification. Doing property based testing well involves thinking carefully about the logical properties that your code relies on and adheres to, which I think is usually a good investment.

Gabriel THOMAS

August 2, 2023 at 8:56 am

Great : each IDE is so much work to learn it. I agree.

---

ok

June 22, 2016 at 1:47 am

I use frameworks like spring-boot. And with dependency injection its sometimes hard to figure out, why i get a result that i did not expect. So i use the debugger to get a feeling, what is called and how that thing has been wired together and where the part actually is, that produces my result. That helps me.

Tomasz Jamroszczak

June 22, 2016 at 7:35 am

When showing disregard to debuggers, people usually make very serious assumptions.

1. Debug printfs are better than debugger. In reality, it's easier to set up ad debug printf inside debugger than inline. So the distinction debug printf vs. debugger is a false one.

2. It's better to know and understand your code. For sure it is, but frequently you just cannot. Imagine code with poor quality, written by strangers, as a quick hacky proof of concept and developed for the next ten years in the same manner. You don't have infinite time to make mental model of the code and finding flow is hard in itself, so then debugger greatly speeds up the process by simply showing call stack and state of local variables. Or imagine implementation of simple graph algorithm – without drawing it on a piece of paper it's hard to grasp what's going on, even if it is short, well structured, with comments and unit tests and functional tests. It's easier to use debugger with conditional statements to figure out what to write with pen and paper.

That's the same argument mathematicians did with regards to mathematical proofs: I don't need no tools other than those from primary school to do maths. And then came four color theorem proof.

3. It's better to have tests. But what if there's a bug without a test for it? In case of Linux kernel it's relatively easy to use sheer power of your mind, especially when you're authoritarian who've seen all the changes. Try to do the same with UI code of Chromium with hundreds of committers and frequent serious redesigns.

What if the tests are unstable? What if the tests started failing because of two different, on a first glance not connected reasons, and one of the reason was introduced into code few months ago in place you're not familiar with?

Again, it's false dichotomy.

4. I own the code, thus I have the obligation to understand it now and for the future. But what if you have to understand minified JavaScript of a random webpage without contacting its authors?

5. In some cases some people don’t need debuggers, thus debuggers are superfluous. Those people are famous, so they have to be right. Typical demagogy.

6. Editor and debugger are distinct environments. Well, no, IDE contains both editor and debugger.

7. I dislike Microsoft so I don’t consider Visual Studio debugger as a decent tool just because.

8. I mainly program in languages which debuggers are inferior or with operating systems on which debuggers are inferior thus all debuggers are unusable.

Mat

June 22, 2016 at 1:34 pm

Thank you!

Daniel Lemire

June 22, 2016 at 1:50 pm

It seems that a lot of your argument is that using tools can make you more productive. I agree with that.

Some comments…

Tools are not neutral. Some systems discourage good practices like clean maintainable code and systematical testing. Getting people to use tools to cope with crappy code instead of having them fix the crappy code is not a net win.

It’s better to have tests. But what if there’s a bug without a test for it?

Then write one. I sure hope you are not trying to fix bugs without systematic testing. It is not 1985 anymore.

I won’t blame you for using a debugger… but I will certainly blame you for working without systematic testing.

Try to do the same with UI code of Chromium with hundreds of committers and frequent serious redesigns.

I sure hope that Chromium is not held together by people running the code line-by-line in a debugger.

Those people are famous, so they have to be right.

That's not my argument. My argument is that if these highly productive programmers do well without debuggers… then it tells us something. Short answer: debuggers are not required to be highly productive. Note that it does not tell us that using a debugger prevents you from being productive. Donald Knuth, for example, uses debuggers.

I dislike Microsoft so I don't consider Visual Studio debugger as a decent tool just because.

I am unclear why Microsoft keep popping up in this thread. I don't think I mention Microsoft in my post. It is irrelevant.

I mainly program in languages which debuggers are inferior or with operating systems on which debuggers are inferior thus all debuggers are unusable.

I am not sure what these languages or systems are. JavaScript used to have poor support for debuggers, maybe that's what you have in mind, but that has changed in recent years. It is fairly easy to use a debugger with JavaScript today.

Damien C.

June 22, 2016 at 8:16 am

I would like to know how those people do to find memory leaks in a monstruous piece of very bad code that has been managed by other people during 10 years (and of course you're unable to use precious things like valgrind and others).

I have a limited use of debuggers but there are cases where they are just usefull.

Frans O.

October 23, 2018 at 3:46 pm

@Damien C.: Valgrind is precious, I agree. If you cannot use that, then — if you are able to reconfigure your build and recompile your code — there are various wrappers for C standard library memory allocation API (malloc, realloc, calloc, strdup, free, maybe others) that help finding memory leaks. GNU C library also allows registering hook functions for those functions (see https://www.gnu.org/software/libc/manual/html_node/Hooks-for-Malloc.html). There seems to be also builtin support for malloc debugging within GNU C library itself (see https://www.gnu.org/software/libc/manual/html_node/Heap-Consistency-Checking.html). With LD_PRELOAD_PATH trick on Linux and perhaps using other methods in other platforms, you could override the malloc functions with your own versions. I do not see how debuggers help you find the location of memory leaks, as for big application there are so much to look for without indications given by these kind of tools.

Gerome Datenblatt

June 22, 2016 at 11:59 am

The article is correctly arguing but wrongly labeled. It is in almost all cases correct not to use a debugger … to single-step through code. But there are plenty of other things debuggers do. The most important one for me may be the ability to add printfs into an executable while it is running or even after it has failed. In case of hard-to-trigger bugs that may save crucial time that is otherwise spend on reproducing the failure. I also often use the debugger as disassembler of the important parts to check wether the compiler actually agreed with me that some abstraction was zero-overhead or some optimization might be a good idea.

That said, most languages come with debuggers worse than those of Turbo Pascal, so spending time to learn to use one may be wasted effort nonetheless.

Tropper

June 22, 2016 at 3:46 pm

I feel bad for your students. I agree that a most simple bugs can be detected just by thinking about what happen, what should happen and how the code looks. And I also agree that there are cases where it is really hard to use a debugger.

But: all in all not using a debugger and saying it is useless is one of the dumbest things i've heard in a long time. I've seen people "not using a debugger" – and if they would knew how to use a debugger properly they could says so much time.

And for the println/assert guys:
– assert are just are poor mens unit test. You wouldn't need them if you had proper units test.
– using logs to monitor a running system is fine. But using println to find a bug on your local system is most inefficient way to do it. And if you teach your student such nonsense they will have a pretty hard time in there professional life.

Igmar Palsenberg

June 29, 2016 at 2:09 pm

Debuggers tend to become decreasingly useful, especially if you have high-volume throughput systems that are full of async code. Debuggers are pretty useless here, since the debugger state usually doesn't match up reality.

We tend to have a lot of error checking and asserts in code, since they tell is we have a real problem. We use debuggers, but mainly as a tool to catch certain asserts (which don't always

directly show the root cause), and as a means to see state at a certain point, without having to dig though a logfile that spits out 4000 lines of debug info for every request.

It's also pretty difficult to unit test bugs that happen to pop up very infrequent, and only occur in certain situations, when certain messages arrive in a certain order. I've had asserts catch more then the unit tests.

Stefano Miccoli

June 29, 2016 at 8:48 pm

In my (limited) experience, unit testing and `asserts` lie on almost orthogonal dimensions: you cannot substitute one for the other. Suppose you are developing a CUDA kernel, say for some linear algebra algorithm: a few well designed `asserts` can catch bugs that are very hard to even notice by unit testing. On the contrary, without unit testing, broken but "formally correct" portions of code do not trigger any assertion failure…

Joe

April 16, 2018 at 8:39 pm

An assert works always, not only on a particular set of inputs.

Roger

June 22, 2016 at 4:02 pm

I can see some of the points you're getting at, but I found the following amusing.

http://imgur.com/PKdkoUE

Vinod Khare

June 22, 2016 at 4:06 pm

I agree with what you say. I tend to use debuggers as glorified print statements. With a debugger, I don't need to write a print statement for every variable I need to watch, all variable values are available when a breakpoint is hit.

One important information that a debugger offers is the stack trace which is very useful especially when debugging event based software. Stack traces are hard to obtain with simple printfs.

Daniel Lemire

June 22, 2016 at 5:07 pm

Stack traces are very useful when you can get them. As Java demonstrates however, it does not require a debugger.

Simon Hardy-Francis

June 22, 2016 at 10:47 pm

I also stopped using a debugger and over 15 years ago. The reason for this is that debuggers are not that useful if the program being debugged is multi-threaded. Instead, logs are useful. And once you start down the path of logging then there's almost no need for the debugger. Plus logs are universal but your favorite debugger feature might be available only in gdb but not in the Windows or embedded system debugger.

I've worked on a few bigger projects which have the printf()s embedded all the time in what could be called 'permanent instrumentation'. In these projects then it was possible to build a debug version of the program which contained all printf()s for all levels of verbosity. And you could also build a non-debug version which had most of the printf()s stripped out. The debug versions also supported special function entry and exit printf()s in order to format the resulting log hierarchically to show something similar to the stack trace but for the entire run-time call tree, i.e. you can see the last assert in the log, see the printf()s before it, see the called function, and all the other called functions before it.

As a side note, I've also developed techniques in the past to automatically instrument C functions with entry and exit printf()s. It's a useful technique to comprehend a new source base faster. For example, I tried it on WebKit which was about 500 MB of source code and just too big to look through! I also have a prototype somewhere for automatically instrumenting projects without changing one line of source code or make file. I remember it worked pretty well on the NGINX and Tor code bases. It worked by pretending to be the compiler (e.g. CC=secretly_instrument make) and then during the compile of the debug version it would secretly generate assembly output, and change that assembly output so that each function had a shadow function and called through a vector. This was a bit clunky and only worked for Intel CPUs, but was an easy way to generate the run-time entry and exit printf()s without doing any work.

Anyway, once the permanent instrumentation is available then there are special benefits, especially in a team environment. For example, if a unit test stops working or becomes flappy, just compare the debug log with the last running version with the debug log of the failing version. 9 times out of 10 it's completely obvious what the problem is after doing the comparison, and this is generally much faster than firing up the debugger and single stepping etc. And if log info is missing then just add more instrumentation. And for new developers

joining a team then the permanent instrumentation gives them a new way of comprehending the code base in addition to reading the source code; they can read the human friendly debug logs and see which function calls which other function and the main flow etc. When you start programming like this then you soon start to realize that the permanent instrumentation just mostly replaces comments, but are way more useful than comments.

It's sucky however when you get used to this technique and then try to use it in higher level languages. Why? Because the more permanent instrumentation is added to a code base then generally the slower it runs at run-time. In C it's possible to work around this issue with the non-debug version using pre-processor macros to ignore printf()s at compile time which have a verbosity which is too high. In this way the non-debug version of the program is not impacted at run-time with constantly executing the equivalent of if($current_verbosity > $this_verbosity){ printf(...); }. Also, the physical size of the source code is not impacted... which can have an effect on performance too. However, most scripting languages do not have an equivalent mechanism and who wants to use the C pre-processor with PHP? Not me 🙂 Plus it's difficult to retro-fit an existing source base with the C pre-processor.

So what can you do? You can use Perl which has a feature called 'constant folding' built in. That means if you write if(MYCONSTANT){} and MYCONSTANT is zero then the entire if() statement never gets compiled. In this way you can add as much permanent instrumentation to a Perl script as desired and if you switch off logging then the Perl script will run as fast as if it had no logging; there is no if() clauses getting constantly executed under the covers.

However, recently I wanted to add permanent instrumentation to PHP. So what to do in this case? PHP doesn't have any equivalent to constant folding, so the more if(verbosity){printf()} statements I add then the slower the PHP scripts run, not to mention that the run-time byte code gets fatter 🙁 In the end I created a simple FUSE file system which duplicates one source tree into another one called the debug tree, e.g. ./my-source/... and ./my-source-debug/... . If a program loads a PHP script from the debug source tree then under the covers the FUSE program loads the source code from the other source tree, but before delivering the source code, it 'uncomments' permanent instrumentation lines. In this way the permanent instrumentation lives in the source code as regular comments and is therefore guaranteed to not cause a performance issues at run-time for the non-debug version of the code.

Using no debugger, permanent instrumentation, unit tests, and enforcing 100% code coverage then it's possible to create programs with remarkably few bugs in them. I recommend this technique.

Anon

June 23, 2016 at 3:46 am

I think the new trend is to write good unit tests with high coverage plus having good logging statements in case something goes wrong and you want to find out where the problem

occurred. The logs are the new standard in big companies for figuring out where the problem happened. This is specially useful in today's big server software where you cannot really step through the code.

Jan Schulz

June 23, 2016 at 9:48 am

I once watched a pro using the Visual Age for Java (that was 2001…) debugger… wow, just wow! I think you could use the debugger in VA to **write** code which was then loaded into the VM and executed. So basically you could step into your empty function, see what you have and then write the next line and step one step below and so on.

I agree with the "think about the design of your code" stuff and using unittests and logging. But regarding prints vs debugger, if you know how to use a debugger, you get a much better overview and are much faster than (iteratively) adding print statements. Especially if you use unittests together with a debugger: you get the bug nicely isolated and can then run through the internals. Usually, when I come to a hard to debug problem, I'm often annoyed that I started with print statements and didn't go the debugger route directly.

[Note: I just used interpreter based languages like java, python, and R, not C and I use only the fancy GUI based debuggers in graphical IDEs]

---

keithy

September 23, 2018 at 3:13 pm

Visual Age for Java got that from the original Visual Age which was Smalltalk, and I remember doing back in 1994. Smalltalk's interactive debugging capabilities goes back to the 1980s. Still the best!

Patrick

June 23, 2016 at 10:46 am

Debuggers are not an alternative to good code design, it's not about "using a debugger" OR "writing good quality code". You can (and should) do both. Tools are here to save time and help you find issues you didn't even think of, they are no substitute to best practices.

You are talking of line-by-line interactive debugging, but that's only a small part of the picture. Modern debuggers can:
– be started non-interactively, without GUI, and generate bug reports
– can rely on data provided by version control

– can be fully integrated within continuous integration tools
– etc.

In my company, we use debuggers. That doesn't mean we waste our time with tools or botch our applications. Quite the opposite. We use them every day without even noticing because we have seamlessly integrated them in our development workflow.

Also, keep in mind that all developers are not experts. Some people (e.g. students) are beginners and debugging tools can really help them understand what they are doing.

Daniel Lemire ▲

June 23, 2016 at 1:46 pm

@Patrick

Elsewhere on this blog, I have repeated promoted the use of better tools. I am definitively not a fan of working on "raw" code as if compilers had just been invented and we had nothing else to go by.

I think that my blog post is clear on what I mean by "debugger" which is interactive line-by-line execution. I specifically say that I use tools that are called debuggers. But I do not use them as debuggers.

Siderite

June 27, 2016 at 1:45 pm

I think your article stands on a semantic issue only. What is a debugger, other than a tool that displays values at certain points in your program? The step by step line execution is not what defines it. Even when software scales to much that it is impossible to step-by-step, as it is in case of parallel programming, you still use a debugger to check states at a certain point. As such, using print statements or placing a breakpoint and then checking what values are at that point seems to be the same thing.

But maybe you are on to something. I would find value in a hybrid approach that just appends bits of code in certain places of a program. With this system pausing the execution and then displaying some user interface would just be a piece of code you appended, but not part of the source code. You want to log the values in some part of your program, you can add logging code in this way. Food for thought.

Daniel Lemire ▲

June 27, 2016 at 2:22 pm

I think your article stands on a semantic issue only.

I don't think it is only semantics. I used to debug my programming with breakpoints and line-by-line execution (15, 30 years ago). I think it is quite common still.

I think that my use of the terminology is common and clear from my post. Here is what Linus wrote:

> *I don't like debuggers. Never have, probably never will. I use gdb all the time, but I tend to use it not as a debugger, but as a disassembler on steroids that you can program.*

As such, using print statements or placing a breakpoint and then checking what values are at that point seems to be the same thing.

Exactly. That is, if you define "using a debugger" broadly enough, then everyone uses a debugger and the statement is void of meaning.

Sergey Vlasov

June 27, 2016 at 6:16 pm

First of all, I totally agree that rethinking code and adding tests/assertions are much more productive in the long run than debugging.

The thing I want to clarify is what exactly makes "stepping line-by-line through code" bad. I think there are three big problems: you need to know beforehand where to place an initial break point, the debugger shows variable values/call stack only for a single point in the program lifetime, and it takes time to get to this point.

"Print statements" are better as they create a log showing program state in multiple points and it is faster than stepping through code manually.

Personally, I rarely use print statements either as there are tools that can show program execution details without manual instrumentation. (Like my Runtime Flow tool for .NET and other similar offerings for Java.)

Preston L. Bannister

June 27, 2016 at 7:57 pm

I am pretty much in the "all-the-above" camp. I use assertions. I build tests. I think carefully about what I write. And I use debuggers.

But not all the code I use is mine, or as carefully done. I do not know all the possible state transitions.

In a prior project, for my code to work, all the other code incorporated in the project had to work. This included code from other developers (who were not as careful), and boatloads third-party libraries. (This was Java code, and the group had unfortunately chosen to use the Spring framework – which is full of unexpected behaviors.)

Using a debugger (in the Eclipse Java IDE) was an enormous productivity boost. When running tests (quite a large set towards the end of the project), an error occurs. Perhaps there is a stack trace (often but not always useful). Using log entries to narrow down the scope of the error, place a breakpoint. (Ideally on a path only taken in the error case. If possible, introduce such a path.) Re-start the test(s).

Sometime later – possibly minutes or hours – the breakpoint is hit, and the IDE is showing the point of error (or near). Often inspection reveals the error, update the code, and Eclipse IDE can do in-place replacement of the code. Resume the test.

When you are using long running tests, and failures are somewhat non-deterministic, this approach is an enormous boost.

That said, I used a debugger much less often on the last project (Python code with OpenStack), and could primarily rely on logging. Depends on the problem.

Daniel Lemire ▲

June 27, 2016 at 8:14 pm

I agree that there are cases where using a debugger with breakpoints is the right thing to do. I also agree that IDEs like Eclipse can be great, sometimes. What you describe does not sound like fun however:

Sometime later â€“ possibly minutes or hours â€“ the breakpoint is hit, and the IDE is showing the point of error (or near). Often inspection reveals the error, update the code, and Eclipse IDE can do in-place replacement of the code. Resume the test.

I think that people like Rob Pike are telling us that there ought to be a better way.

---

serhan iskander

June 30, 2016 at 11:06 am

if you work alone “Maybe”,
but working in any company you will have code that existed for a long time, and even you will

forget your code, debugger help you pinpoint the problem much faster.
you need to write good code and know where the problem is but it's extra tool than can help and your as good as your tools.

Daniel Lemire

June 30, 2016 at 2:06 pm

> *your as good as your tools*
>
> I do believe that there is a lot of truth in this.
>
> *you will have code that existed for a long time, and even you will forget your code*
>
> Linus Torvalds does not use a debugger. Do you have code that is qualitatively larger, older and more complex than the Linux kernel?
>
> *if you work alone Maybe*
>
> There are at least 300 developers involved in any new version of the Linux kernel. How big is your team?

Scott Blum

July 1, 2016 at 4:43 pm

Agree with the points others have raised. Even as someone with a bit of a reputation among my colleagues for spotting bugs via inspection a on code review (and also writing understandable code myself), I still find a debugger so valuable and time-saving on occasion that I tend to avoid environments where running a debugger is impossible.

The central problem is, no matter how good your own designs and code are, developers are not islands anymore. In any project of a reasonable size, your own code is only a tiny fraction of what's happening. The rest of it is your coworkers, the language's core libraries, and numerous third-party libraries of varying quality– both in terms of correctness and how easy the code is to read. The former you can't change, and for the latter, you might submit a fix for a third party lib, but you're probably not going to investing in cleaning up the architecture. There's just no substitute for stepping through JRE classes which something really unexpected is happening.

Lukas Eder

July 2, 2016 at 2:27 pm

Interestingly, those people you cite probably hardly ever work on other people's code (or other people's libraries).

For the rest of us, debuggers are excellent tools for figuring out how the legacy software that we're maintaining works, or to report bugs in third party libraries.

Rainan Miranda de Jesus

March 19, 2019 at 12:52 pm

Linus Torvalds have about 300 developers working in Linux kernel with him... I think you might rethink your comment.

JosÃ© Duarte

July 7, 2016 at 11:36 am

Linus' coding practices are not to be commended, especially after the recent Git security vulnerabilities, which were completely predicable given the software development process that is typical in the Unix/C orbit. There's a certain steampunk, almost Luddite, approach to software in Unix, and the debuggers are pretty bad.

The reason Microsoft keeps coming up here is that Visual Studio's C++ debugger is widely regarded as the best in the world. (I'd expect the same to be the case for their C# debugger, but they have less competition there.)

Putting together some recent threads, here and elsewhere, I think there's a real deficit in the academic CS community, a stunning lack of awareness of large swaths of present day computing. There's little awareness of Microsoft tools, Windows, and related file systems, languages, and so forth. This is compromising the quality of CS research – the community is far homogeneous and conformist with this whole Unix and C thing. And it leads to things like advice to not use debuggers, based only on this steampunk Unix/C paradigm. I feel like this monoculture slows progress in computing. Separately, software security is devastating right now, and we need much more powerful tools to deal with it than C and its printf flintstones. We need much richer debuggers with all sorts of modeling methods.

Daniel Lemire ▲

July 7, 2016 at 2:05 pm

Linus' coding practices are not to be commended, especially after the recent Git security vulnerabilities, which were completely predicable given the software development process that is typical in the Unix/C orbit.

There are more computers running the Linux kernel than computers running any other operating system. Linux is the foundation of our Internet…

Putting together some recent threads, here and elsewhere, I think there's a real deficit in the academic CS community, a stunning lack of awareness of large swaths of present day computing. There's little awareness of Microsoft tools, Windows, and related file systems, languages, and so forth.

I think that the vast majority of CS academics are Windows users. The vast majority of CS courses are prepared on Windows for Windows users. Probably, in second, you find Apple technology followed, far below, by Linux technology. Though I use Linux machines (as servers), I haven't used Linux as my main machine in a decade or so. I know a few exceptional CS professors who use Linux as their primary machine… but they are few.

It is rather in industry that you find the most massive Linux adoption… at key corporations like Google, Amazon and so forth. Academics are not behind the sustained Linux popularity. In fact, if you go to a random CS school and pick a random CS professor, chances are that he won't be able to tell you much about modern Linux development tools. Most academics have had nothing to do with cloud computing and have never setup a container, assuming they know what it is.

Andy

December 16, 2020 at 8:13 pm

> *Visual Studio's C++ debugger is widely regarded as the best in the world*

Just visiting to LOL at this!

Daniel Lemire ▲

December 16, 2020 at 8:14 pm

Please elaborate.

Anselmo

July 21, 2016 at 7:14 pm

Great post! As an engineer, I always thought that using prints to debug code was an amateur thing. I hardly use debuggers, only when I cannot figure out the problem by investigating the prints.

Greetings from Brazil.

Ashish Hanwadikar

December 17, 2016 at 8:47 pm

Agree 100%. I made the same point in an linkedin post (https://www.linkedin.com/pulse/interactive-debugger-considered-harmful-ashish-hanwadikar).

Dirk Schuster

January 11, 2017 at 3:26 pm

Thank you!

Just stumbled upon the sentence “step-debugging is one of the key skills for any developer” somewhere and wanted to see whether there is anyone else but me who is missing this “key skill” and so arrived at your post.

My history is quite similar to your’s (Basic at school back in the 70ies, Turbo Pascal, then C++ and now mostly Java).

Very much like you said I actually stopped using a debugger when I (or rather my code) started to get seriously multi-threaded. For debugging I use assertions, sometimes only “soft assertions”, ie. simply error-logging when some expected condition does not hold, and logging/tracing. And yes, I frequently read and work on code produced by others, in particular when taking over after someone has left the company. To understand other’s code ‘grep’ and ‘etags’ are much more helpful tools for me than a debugger.

Stepping, break-points and watches usually don’t provide more information than logging statements can do, but logging statements provide a history of the program execution and are still there when some error occurs at later stages of the program’s life cycle, ie. in production. When talking about logging/tracing I refer to configurable logging tools not printing to stdout (like log4j/slf4j rather than System.out.println in context of Java).

Actually I once was in a situation where even logging didn’t help, because the error disappeared whenever even the least logging statement was inserted — obviously a tight race condition. In this case there was no way other than debugging in mind and it actually took me two days of mental stepping through the code to find the bug.

Currently I only use a debugger when debugging Lisp (Elisp) code but then exactly as you analyzed it is a single-threaded situation and small debugging objects (code units with at most a few functions).

Razvan

February 3, 2017 at 7:02 pm

I disagree. One says that a debugger doesn't scale, but logging (print is a primitive form of logging) does. Now, your 15 million lines code would be only 10 million without extensive logging you had to do because of lacking debugger. Plus, searching something in the agglomeration of messages turns not scaleable too.

To me, debugger is a great tool, by providing the capability of inspecting at once, in a given moment, all the local variables, stack or the members of a class.

In my career I've debugged C, C++, -O3 C++, Java, Pyton and Assembler. If an IDE doesn't have debugger, it does not qualify for my attention. Not knowing to use a debugger (as well as a profiler) is a red flag.

Daniel Lemire ▲

February 3, 2017 at 8:24 pm

> *One says that a debugger doesn't scale, but logging (print is a primitive form of logging) does. Now, your 15 million lines code would be only 10 million without extensive logging you had to do because of lacking debugger.*

Must debuggers and logging be our primary tools to debug problems?

---

asm123

June 12, 2017 at 6:43 pm

15 millions of code lines for a kernel sounds like the most mismanaged project in the Solar System. I wouldn't give a cent for that code, the binaries nor 100 hours of the one managing such nonsense.

Andrew Norris

November 5, 2017 at 6:10 am

The advantage of print statements is you are printing the key data you need to verify.

Often code goes wrong more than once. Esp. when first making it but can happen much later on.

So it goes wrong again at some date in the future : you have all the key results printed out for you. This both helps you see how it works and also at what point it stops working.

I usually just rem out the print statements once the code is running so I can come back to them later if the code is not working right.

Profilers, however are quite a different story than debuggers. Very useful !

Herbert

January 24, 2018 at 12:43 am

It all depends on the problem, for debugging GUI apps, using print statements is not always viable. Sometimes it's simply not possible to use a debugger and print statements are the only thing possible for example in multithreaded apps. It's foolish to say that one doesn't use a debugger just because. Stack tracing alone in a complex piece of code is really useful especially is one is not familiar with the code. What's most likely happening is that those who don't want to use debuggers have probably never used a modern debugger and don't see the productivity benefits. When I refer to a debugger I'm thinking of those that come with Visual Studio or the Delphi IDE not the command line debuggers like GDB, there is a world of difference between the two. Perhaps this is why Linus doesn't use debuggers simply because there weren't any truly modern debuggers when he wrote Linux.

Daniel Lemire ▲

January 24, 2018 at 1:53 am

It's foolish to say that one doesn't use a debugger just because.

Except that I don't write "just because".

Stack tracing alone in a complex piece of code is really useful especially is one is not familiar with the code.

I include this as an exception in my post.

I'm thinking of those that come with Visual Studio or the Delphi IDE not the command line debuggers like GDB, there is a world of difference to between the two

As my post makes clear, it is hardly "modern" to have a graphical user interface for your debugger. It existed well before Linus built Linux.

Perhaps this is why Linus doesn't use debuggers simply because there weren't any truly modern debuggers when he wrote Linux.

If you use inferior tools, you will be outcompeted by others. If you have massively superior tools than Linus, then I am sure you are producing much better software. Right? Are you confident about that?

Bill

October 17, 2018 at 6:48 am

One tool not mentioned for debugging is dtrace, ftrace. The ability to turn debug paths on and off dynamically and to target the debug to narrow portions of operation pretty much gives you an interactive logging retargetting approach which would be very difficult in a typical debugger no matter how you used it.

When debugging infrequent non deterministic bugs this is quite a bit more likely to expose both the bug and the path leading to it. Scheduling entities are more essily captured.

The minimal hit to timing is not possible with any globally aware debugger.

This turns out to be the more productive path for me

---

wqweto

October 20, 2018 at 12:50 pm

Not using a debugger is like not using a linter. Your project is not going to die but usually the debugger gives you a new perspective as to what this monster of code you've created is doing behind the scenes.

Linus might not need a debugger but are you as good as him to claim you understand all of your codebase and no more insights are needed?

My point is: Be humble and use a debugger, man! You are not that much better than the schoolboy that dabbled with those Borland's awesome debuggers. (You remember Ctrl+F2 to evaluate *any* expression with local and global variables at run-time with no recompilation?)

Daniel Lemire ▲

October 20, 2018 at 2:32 pm

I love linters!

And no, I am not much better but I work on much harder problems with more sophisticated tools. Conventional debuggers just don't scale to billions of function calls and tens of threads.

My optimized code gets compiled to something that barely looks like my source code. I use gigabytes of memory not 512 bytes.

---

icsa

May 4, 2019 at 11:59 pm

I'm definitely in the "I don't use a debugger." camp. With that said, I've used some extraordinary debuggers including the Jasik Debugger for the Macintosh.

However, when I went to use a debugger after having inserted print statements, I always came to one or more of the same conclusions:
* I didn't understand the program
* I didn't understand the problem the program was supposed to solve
* I didn't understand the programming language used to solve the problem.

Once I made sure that I understood all of the above, the need for a debugger went away.

---

icsa

May 5, 2019 at 12:00 am

I do still use print statements or logging to show the state of a running program.

---

adampasz

May 5, 2019 at 12:02 am

Debuggers are like microscopes. They are great for untangling complicated problems at the function level. They are not so useful for understanding interactions between components, or for dealing with asynchronous events.
Lately, I've been using trepan and vim for Node development. I use tmux to switch back and forth, with very little impact or slowdown from context switching. I also like running my code frequently while I'm writing it.
This is what works for me. I've found when I use debuggers less, I end up adding more logging which is merely an approximation of what I could see in the debugger, and adds a lot of clutter to my work, and slows me down. I get that, once you deploy, you can't always rely on debuggers. This is the point when I think about adding well-crafted logging.
Different programmers have different approaches. That's OK.

Sameer Patil

May 5, 2019 at 4:35 am

Interesting blog. Your arguments are valid. But if I don't use GPS navigation, I would know the roads better. If I don't use calculator, I would be better at mental arithmetic. Well, productivity is also important.

Daniel Lemire

May 6, 2019 at 1:29 pm

I am not advocating using lesser tools. I am pointing out that debuggers are limited and may impair productivity.

Feroze Daud

May 5, 2019 at 7:50 pm

Daniel, I think you make some good points. And I know you are not really saying nobody should ever use a debugger. However…

1) No matter how much clean code I write, with unit tests and all, I cannot cover everything.

2) Writing printfs only helps when you have an inkling of where the bug might be, and you have the luxury of being able to repro it on your machine.

3) This works fine in SAAS/PAAS scenarios where no product is being shipped to the end user.

4) When you have products being shipped to the end user, and you get crash reports or other telemetry indicating problems, you cannot go back and write printf or log statements. Even if you did and shipped an update to the customer, it is not necessary that he will be able to repro it.

5) Of course, reproes are not guaranteed in the debugger too… but atleast you can try complex things, like seeing if thread races are causing problems, by pausing a thread in debugger and letting others continue.

6) A class of bugs, eg, off-by-1 bit errors due to memory problems, access violations etc, esp in the presence of no symbols, are sometimes only invetigatable using a debugger.

Daniel Lemire

May 6, 2019 at 1:12 pm

Of course, I do not think we should never use a debugger, but your comment suggests you might be advocating something different.

1. Testing cannot cover everything but the time you spend in the debugger is time you are not investing on testing.

2. If you do not have access to the machine, how do you run a debugger on it? Are you mixing up the concept of a debugger (a manual tool the programmer use) with the build settings? Releasing the code with bound checking is entirely different from using a debugger.

3. If the product is getting shipped, then you should prioritize testing and logging over debuggers since you are not going to be able to use a debugger on the running code.

4. See my 3.

5. and 6. There are definitely automated techniques to detect data races and access violations. If people think that the only tool for these problems are manual debuggers, then they are wrong. We have had checkers and sanitizers for decades. Fixing the problems is still manual labor, of course.

Feroze daud

May 7, 2019 at 12:43 am

Ok now imagine I am a developer at Microsoft. Everyday I get crash dumps sure to my Software crashing users machines. How do I find out where the problem might be?

Daniel Lemire ▲

May 7, 2019 at 1:10 am

Debugging from core dump is not what I would typically think of as “using a debugger” though it may involve that.

Eric

May 7, 2019 at 12:33 am

I don’t really understand this. I agree that gray-matter debugging is extremely useful, but for any moderately complex project there are bound to be third-party libraries brought in. Sometimes these can do unexpected things that debuggers are great at catching.

Unless it’s an outdated or terrible debugger, modern debuggers are fantastic at showing the behavior of locking primitives and the interweaving of async code. I use Visual Studio, which allows me to see the call stack of each thread in my process and then step back in time and play

"what-if" scenarios with the outputs of various modules. By being able to see why each thread is where it is at and playing with the data that comes back from functions in the stack, I can quickly narrow an issue down to something in my code or an unexpected side effect in a library we brought in.

As a side note, we adopted F# as our primary language (we're mostly a .NET shop). It was not an easy transition since OOP has been banged into our heads for the last 20 years, but thinking about problems as a flow of data through idempotent functions rather than the evolution of system state has made our code much easier to reason about mentally. As good as we are at having automated tests and continuous integration, we still sometimes catch bugs in the debugger that would have been gnarly to catch with printf's. The quickest way to narrow these types of things down is to set up explicit break conditions for things that shouldn't happen and inspect the runtime history of the threads involved if they do.

Debuggers are not a panacea but they are a useful tool for projects that have at least a moderate integration topology.

Daniel Lemire

May 7, 2019 at 12:59 am

I specifically allude to this use case… *there are cases where a debugger is the right tool (…) in contexts where my brain is overwhelmed because I do not fully master the language or the code.*

Feroze Daud

May 9, 2019 at 8:41 pm

Definitely, good coding practices help lessen the need for debuggers but do not eliminate them.

Grey matter debugging works fine if you have the luxury of working on the same problem for a long time, without distractions, or if the code is something you wrote or is simple to grok. For all other real world applications, it does not work…

But there does come a time if you are firing up the debugger frequently to debug a piece of code, where you have to think if it would be better to write more unit tests instead. But again that works if it is your own code, or component on which you can make changes.

---

fdf

June 7, 2019 at 11:20 am

R studio is terrible. Because you dont like debuggers it does not mean that people should not debug. sorry

R is in general quite a weird structure scripting language. I think its complicate manner makes people feel its a language.

Carla Sanchez

June 22, 2019 at 5:51 pm

"…I was able to find several famous programmers who took positions against debuggers…"

That's when I stopped reading.
https://en.wikipedia.org/wiki/Argument_from_authority

Never trust an argument that starts with a fallacy, particularly one that is common enough to have a name.

Daniel Lemire ▲

June 24, 2019 at 1:48 pm

Stating that many people do important work without debuggers is not a fallacy. It is a fact.

---

seebs

June 27, 2019 at 7:36 pm

I use debuggers raarely enough that I don't actually know whether I *have* a debugger for the language I'm currently working in most. After a great deal of watching other people debugging, I've concluded that, on modern systems, a debugger is almost always *slower* than studying carefully-considered logs, with the key exceptions being mostly memory corruption. For nearly everything else, being able to see a series of logs over time seems to work better.

In particular, I think I get a lot of benefit from *thinking* about what I want the logs to tell me.

Luis Goncalves

July 16, 2019 at 5:41 am

There's one thing that most debuggers/languages don't have (except for Matlab) that would allow debugging to scale : "dbstop if error". That means if an error occurs, the program doesn't crash and exit, but it stops right there into debug mode, where you can inspect what happened without having to set a breakpoint and re-run the code again to recreate the error. (In matlab that means you get the same fully functional interpreter prompt that you started with (unlike python's pdb!), and can do anything you want (plot, call functions, compute, as well as move up and down the calling stack to understand how you got to the error).)

The concept of debugging on error with a fully functional prompt may seem trivial and not much of an improvement, but it actually is quite significant, because it speeds up the debug cycle immensely. I'm both saddened and frustrated that python and julia don't offer something equivalent. It would be a significant improvement to both languages.

Daniel Lemire

July 16, 2019 at 11:57 pm

Can you elaborate?

it stops right there into debug mode

Would you want this to happen in production? I think that, in production, you want a clean crash.

If this is not in production, then surely you can run your script in debug mode, right?

Joe O'Leary

July 22, 2019 at 5:13 pm

I don't know about Luis but I know that I would certainly want to be able make this happen in a production build. In fact, I do it all the time. Some obscure crash is occurring, particularly those that happen at an indeterminate time. Perhaps it is some specific std::exception derivative being thrown. On Windows, perhaps it is some hardware exception like an access violation.

The ability to run a production build in the debugger with the debugger set to break at the exact moment such an exception occurs is invaluable. I don't need to go in and start inserting printfs everywhere. I just see a snapshot live, as it is happening.

Regarding your response to Feroze above, I don't know if you've ever dealt with production crash files on Windows (or if Linux even has an equivalent) but on Windows the process is something like this:

Customer reports application crash in the field.
Customer sends log files back to us along with a crash dump file back to us (which we dump as a last resort)
Usually the log files make the problem clear. But no matter how good you are or how many development/testing resources you have, eventually they wont.
In that case I take the dump file and open it up in the debugger along with the snapshot build of that specific release.
Debugger gives me a live call stack of the crash. I can examine the data live. I can switch

back up the call stack and look at data in those functions. Obviously much is optimized away and I cannot examine it live. But a some of it is not. And what is notI can examine in raw memory
Furthermore the debugger gives me the call stack of every other thread exactly it was running at that time. I can switch to other threads, examine live data in them as well.

In my experience, the ability to do this is a huge time saver. It can be often the difference between finding a problem immediately and finding it days/weeks/months down the line. Especially in the 3-person development shop scenario. Or the “I-inherited-this-godawful-codebase” scenario

---

Daniel Lemire

July 22, 2019 at 5:57 pm

@Joe

> *(…) I would certainly want to be able make this happen in a production build. In fact, I do it all the time. Some obscure crash is occurring, particularly those that happen at an indeterminate time. (…) The ability to run a production build in the debugger with the debugger set to break at the exact moment such an exception occurs is invaluable.*

A friend of mine teaches computer security and he has a whole 3-hour lesson on how to exploit debuggers running in production. Microsoft says that debugging should be disabled on production machine. Admittedly, maybe the concerns are overblown and it is fine to run your production servers in debug mode.

> *Regarding your response to Feroze above, I don’t know if you’ve ever dealt with production crash files on Windows (or if Linux even has an equivalent) but on Windows the process is something like this: (…)*

Yes. I am familiar with it, and yes, as far as I can tell, all major operating systems have core dumps. It should since it is a technology from the 1950s (according to Wikipedia).

Joe O'Leary

July 22, 2019 at 6:20 pm

> *A friend of mine teaches computer security and he has a whole 3-hour lesson on how to exploit debuggers running in production. Microsoft*

> *says that debugging should be disabled on production machine. Admittedly, maybe the concerns are overblown and it is fine to run your production servers in debug mode.*

To be clear getting a crash file does not require enabling debugging on a client's machine. We don't send them the debug symbols or anything.

> *Yes. I am familiar with it, and yes, as far as I can tell, all major operating systems have core dumps. It should since it is a technology from the 1950s (according to Wikipedia).*

Oh yes, of course. I wasn't implying that they didn't have crash dumps. I'm saying that a debugger is an invaluable tool for examining one

---

Or Weis

July 30, 2019 at 8:49 am

How about trying a modern debugging/datapoint extraction solution that scales?

Like Rookout

Ian Newson

October 5, 2019 at 7:51 pm

I don't understand this post.

None of your reasons are reasons not to use a debugger, merely descriptions of situations where they're not the best tool. There are plenty more of those types of systems, such as when you're trying to debug a real time system, but that's not a reason to use debuggers in situations where they are appropriate. Debuggers aren't my first port of call either, I'll typically start by reading the code and trying to imagine states which could produce the issue I'm seeing. Debuggers aren't my last port of call either, it'd be silly to skip over a debugger before jumping to more intensive forms of testing such as creating and analysing memory dumps, or building extra tools yourself.

I guess not even CS professors are immune to the allure of a click bait headline!

Daniel Lemire ▲

October 5, 2019 at 8:54 pm

I merely describe how I work, pointing out that other credible programmers share the same approach.

I write pretty complicated code and I really never use a debugger for debugging.

It is a statement of fact.

That I am a professor has nothing to do with it.

Ian Newson

October 5, 2019 at 9:11 pm

You stated in the post that you do use debuggers, that's the main reason the post is click bait. The title is the only reason I clicked.

Honestly it doesn't matter, of course you should work in the way that benefits you most. The click bait headline is really the only part I take exception to. It's unhelpful and I'm afraid of the possibility that it helps produce programmers who can't or don't see the value in a debugger.

In a while crocodile!

Daniel Lemire ▲

October 5, 2019 at 9:19 pm

It is not clickbait. I am openly discouraging reliance on debuggers.

Ian Newson

October 5, 2019 at 10:21 pm

I do not use a debugger

.

```
there are cases where a debugger is the right tool
```

.

```
I used a debugger ... to debug a dirty piece of JavaScript.
```

These quotes from your article show that you do use debuggers, in defiance of your post title. Therefore the title is a terminalogical inexactitude.

I don't care if you want to discourage the use of a debugger, I'm certainly amenable to that notion, but to suggest it shouldn't be in a programmers toolbelt is foolish and at odds with the content of your post.

Daniel Lemire

October 5, 2019 at 10:48 pm

There few absolutes in programming: it is hard to find practices that one can really never have to do.

But as a general rule, I do not program using a debugger.

I also do not drink my coffee with sugar.

I do not eat desert.

I do not memorize phone numbers.

Etc.

Does it mean that there can be no exception? Of course, there can be.

I would also say that I don't use assembly when programming… but you'll find exceptions.

I would also say that I don't program in Rust or D or Haskell, but you'll find code in these languages on this blog.

Note the exception about JavaScript: I specifically refer to dirty code.

So I stand by my statement: I do not use a debugger… Not as an absolutist religious statement but as a method of programming.

---

feroze Daud

October 6, 2019 at 5:06 am

I can grant you that a debugger should never be the first tool you go to. But if you say that you never used a debugger, then maybe you are not facing the situations where it is indispensable, and in some situations the only one that makes sense.

When I worked in microsoft, the saying was that you formulate why a system crashed – you try to examine the code and come up with a hypothesis, then use the debugger to prove or disprove

it. Just blindly going to a debugger is not going to help.

And as I said, when you are shipping software that runs on end users machines, and then crashes and you get a core dump, you dont have the luxury of looking at the code always. You have to look at the dump along with the other metadata in it (users OS, patch level etc) and try to figure out what happened.

Can you get by without ever using a debugger for the code you write ? Sure. But try doing that when you are part of an organization shipping something complicated like windows, linux, office, sql server etc. And you will not get very far with that approach. You just wont have the bandwidth to read everybody's code and do mental debugging.

Daniel Lemire ▲

October 6, 2019 at 4:00 pm

Linus ships something as complicated as the Linux kernel and he does not use a debugger. Python was built without relying on a debugger. Go was built without relying on a debugger. And so forth.

So the story where you get away without relying on a debugger because you are building simple things does not work. Lots of people who build widely deploying, difficult software do not rely on debuggers.

In some sense, as I have argued, the opposite is true: for many hard problems, the debugger is not helpful. Debuggers are certainly fine to debug throw-away code where you don't want to invest in long-term maintainability, you just want to get going.

Now, I have very clearly, indicated that I do believe that debuggers can be useful, and they have their place. It is certainly the case that if you have to debug code that you do not understand, that was built in a less than optimal way… then a debugger can be a life saver.

Yet, to me, this is like saying that firemen are life savers. They are. But we don't maintain cities by relying on firemen. We make sure that we rarely need firemen.

My main beef is with the debugger-oriented approach to programming which often means few to no unit tests, no assertions, no fuzz testing, messy code architectures, missing or incomplete documenation. I think you should almost always program in such a way that a debugger won't be needed. You should use the time you would spend in the debugger writing code that can be maintained without a debugger.

It is far better to improve your testing, for example, than to spend time in a debugger.

Of course, if you work for a company, you may not get a choice. Maybe your employer won't let you spend 3 months writing tests for your code instead of producing a new feature. And maybe this makes sense because the code will get thrown away in 3 weeks anyhow…

---

feroze Daud

October 7, 2019 at 2:05 am

But again you are making a false dichotomy. The choice is not between debugger and logging/unit testing etc. The choice should be both. Absolutely fortify your product at development / testing time with uni tests, log messages etc. And if that results in bug free product, so be it.

Also.. I dont understand when you give examples about python/go/linux etc. Are you saying that they were developed without using debuggers at all? Or are you saying that they do not ship with debuggers because they dont think end users are served by them?

Again, I dont know the details but I find it really hard to swallow that python /linux were built without ever relying on debuggers. Maybe Linus can pull it off. But what about so many other contributors ? Debugging is kind of like a first class citizen in python. Just "import pdb;pdb.set_trace()" and you are set.

This argument is kind of like a homeopath and MD arguing. Can the homeopath claim that he has cured people. Sure. Can that claim be sustained with the general population? Very few would say "I have never used a doctor"

Debuggers are like that. Should you write your code so that the choice of debugger is rare? Yes! But saying that nobosdy should ever need to use it is far fetched. As they say YMMV. If you can do without one, well and good.

Daniel Lemire ▲

October 7, 2019 at 2:40 pm

> *But again you are making a false dichotomy.*

I don't think I am. Your time is finite. The time you spend in a debugger is time you are not spending doing other work.

If you are building software by relying on a debugger, then the time you spend in the debugger, going line by line or setting up watches or whatever, is time you are not spending writing code, documentation, and so forth.

Now, if you are rarely using a debugger, to the point where the time spend in a debugger is virtually zero, then you are, for all practical purposes, in complete agreement with my post.

> *I dont understand when you give examples about python/go/linux etc. Are you saying that they were developed without using debuggers at all?*

Linus in particular does not use a debugger. Many famous programmers who built the infrastructure you built are in the same boat. Please follow the links I offered in my post.

Maybe you find it incredible that people can write sophisticated, complex real-world software without a debugger. But it happens.

I write hard-to-write software myself, and lots of people are relying on some of it. I'm a not a naive programmer. I write pretty difficult code, all the time. And I don't rely on a debugger while I am doing it.

I know how to use debuggers, that's how I started to debug non-trivial code many years ago, but over time I stopped doing it. Exceptionally, I will use a debugger, but that's usually a sign that something is wrong: either the code is poor or my understanding is poor. When the code is good and I am on top of it, I don't need a debugger.

> *Should you write your code so that the choice of debugger is rare? Yes! But saying that nobosdy should ever need to use it is far fetched.*

I very specifically and very carefully point out that I am not saying you should never use a debugger. Even just in my reply to you, just now, I wrote: *I have very clearly, indicated that I do believe that debuggers can be useful*.

What I am claiming is that you are routinely going to the debugger, then it is probably a sign that you should improve your methods. The debugger should be a last resort tool…

You can reasonably disagree with me, of course. You can believe that using a debugger frequently is sane and productive.

Stevek

July 23, 2020 at 10:51 pm

I totally agree. I used to use a debugger extensively. Then I changed my development environment to always use optimization when compiling, to match production compiler flags.

This makes the debugger not really useable.

But I found that finding bugs by print statements much more efficient. Often just thinking where to put print statements will have you see the issue. It is quite quick.

Peter Guo

August 19, 2021 at 6:14 am

It is better to know what's happening with a debugger.

Arshad T. McGillicuddy III, BSc.

December 20, 2021 at 8:38 pm

I'm perceiving an interesting logger-vs.-debugger dichotomy I wasn't aware of.

I arrived here while searching for comments about logging. In my experience, logs are mostly useless and a minor tool at best. A debugger (however one uses it) is, to me at least, almost indispensable.

And yet most developers will swear up and down that logs are fantastically useful, that they use them all the time, etc. and conversely, we have this phenomenon of people claiming they don't even use a debugger!

I don't know if they're doing it wrong, or if I am, or if it's just a legitimate difference of mere style. I do know, though, that if you watch a developer work you'll sometimes observe things that are at odds with how he *says* he works. For example, I know that some of the people who extol the virtues of logging to me simply don't open up log files very often.

Do Torvalds, etc. surreptitiously step through code in GDB like closeted preachers sneaking to the "wrong" end of Bourbon Street? I don't know, but if you remove the typical bombast from his statement I think he's saying that he's smart and knowledgeable enough to do better than just stepping through code slack-jawed until the light bulb of ideation turns on, and I believe him… usually.

Daniel Lemire ▲

December 20, 2021 at 8:53 pm

> *I know that some of the people who extol the virtues of logging to me simply don't open up log files very often.*

I would argue that if they are doing it often, they are doing it wrong.

Shane Dalton

June 17, 2022 at 12:01 am

I think stating that a few talented programmers write code without a debugger and therefore debuggers are inferior in some way is not a statistically accurate representation of the value of not using or using a debugger. Consider the converse, I wonder how many 'regular' programmers do important work with debuggers that keep our information systems alive? Would it not be the case that the debugger provides a greater net utility to software development? Economics overrides idealism in this argument I believe.

The debugger is faster for a lot of things; though certainly not all things. An example is when I write new code I write unit tests that fail and in the process of their execution I then trigger a break point in my code, at this point I open my IDE's evaluate expression box and write a bunch of code, which I can do incorrectly a large number of times with no time penalty (or a lower time penalty than staring intently at my code and wasting my mental energy to manually execute it in my head which is finite) and when it returns the result I need for the next step of the function/process I just paste the code into my function and move on to the next piece and move my breakpoint as well as restart the test, while !done repeat. In this way I encounter silly syntax mistakes in the debugger and not after a 15-45 second write/test/debug cycle with print statements, and this probably saves days of my life over a year. Many times I will write large sections of my code without running the debugger, but when it fails modifications are much faster in the REPL of the debugger.

I do use print statements/logging to give me a general idea of the state of the program before it failed, but this is just to optimize the time it takes to zero in on the correct breakpoint locations.

I also do write Python which is a lot more dynamic I can import objects in the repl and manipulate as well as explore them (and there are many many objects with many attributes/methods each in a decent sized codebase), and I think for a language like C where the individual context of the point where the debugger freezes is more limited it might be kind of useless and similar to reading a dictionary with a microscope. With python I can hop through the entire call stack, modify things dynamically at run time and inspect the system at a level that's just not possible with printf, assert, log, etc.

Maybe if I was writing mathematical libraries the utility of a debugger would be less useful, but I would write those with a pen and paper first and then put them in code.

I think a balanced diet of logging, unit testing, and debugging is the best option for the work I do (and I believe I represent the general population of people who write software, that is what my job title says, although I may be a bit junior on the CV). Avoiding or ignoring any of those three would be foolish in that it would waste more time and I need my time to be productive

and maintain the security of my job, perhaps that is the difference in our perspectives on this element.

I do value your insight and the general message you are trying to portray (or maybe it's just clickbait ;) of not relying on a single item to the extent that it becomes a crutch, but I think the opposite is also true, in that excluding a valuable tool on dogmatism alone is inviting inefficiency and is also a crutch.

Thanks for the time it took to write this and I will be keeping an eye on how to better incorporate logging in my work in the future.

Daniel Lemire

June 17, 2022 at 1:54 pm

Thanks. It is not clickbait. I genuinely do not use debuggers "to debug" good code that I write and maintain. I do use debuggers for other purposes. And I might use debuggers for other people's code. Debugging with a debugger would be a "last resort".

Pingback: Tracing in Rails – Beyond Velocity
