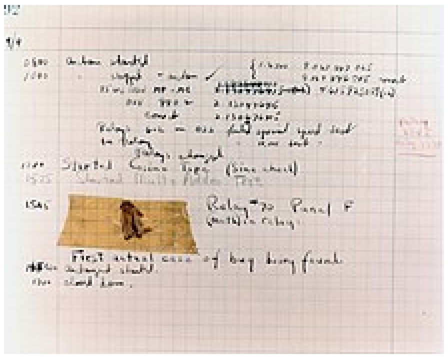
Wikipedia

Debugging

In computer programming and software development, **debugging** is the process of finding and resolving *bugs* (defects or problems that prevent correct operation) within computer programs, software, or systems.

Debugging tactics can involve interactive debugging, control flow analysis, unit testing, integration testing, log file analysis, monitoring at the application or system level, memory dumps, and profiling. Many programming languages and software development tools also offer programs to aid in debugging, known as *debuggers*.

Etymology


[illegible]
1545 Relay #70 Panel F (moth) in relay.
First actual case of bug being found.
[illegible]

A computer log entry from the Mark II, with a moth taped to the page

The terms "bug" and "debugging" are popularly attributed to Admiral Grace Hopper in the 1940s.[1] While she was working on a Mark II computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system. However, the term "bug", in the sense of "technical error", dates back at least to 1878 and Thomas Edison (see software bug for a full discussion). Similarly, the term "debugging" seems to have been used as a term in aeronautics before entering the world of computers. Indeed, in an interview Grace Hopper remarked that she was not coining the term. The moth fit the already existing terminology, so it was saved. A letter from J. Robert Oppenheimer (director of the WWII atomic bomb Manhattan Project at Los Alamos, New Mexico) used the term in a letter to Dr. Ernest Lawrence at UC Berkeley, dated October 27, 1944,[2] regarding the recruitment of additional technical staff.

The Oxford English Dictionary entry for "debug" quotes the term "debugging" used in reference to airplane engine testing in a 1945 article in the Journal of the Royal Aeronautical Society. An article in "Airforce" (June 1945 p. 50) also refers to debugging, this time of aircraft cameras. Hopper's bug was found on September 9, 1947. Computer programmers did not adopt the term until the early 1950s. The seminal article by Gill[3] in 1951 is the earliest in-depth discussion of programming errors, but it does not use the term "bug" or "debugging". In the ACM's digital library, the term "debugging" is first used in three papers from 1952 ACM National Meetings.[4][5][6] Two of the three use the term in quotation marks. By 1963 "debugging" was a common-enough term to be mentioned in passing without explanation on page 1 of the CTSS manual.[7]

Peggy A. Kidwell's article *Stalking the Elusive Computer Bug*[8] discusses the etymology of "bug" and "debug" in greater detail.

Scope

As software and electronic systems have become generally more complex, the various common debugging techniques have expanded with more methods to detect anomalies, assess impact, and schedule software patches or full updates to a system. The words "anomaly" and "discrepancy" can be used, as being more neutral terms, to avoid the words "error" and "defect" or "bug" where there might be an implication that all so-called *errors*, *defects* or *bugs* must be fixed (at all costs). Instead, an impact assessment can be made to determine if changes to remove an *anomaly* (or *discrepancy*) would be cost-effective for the system, or perhaps a scheduled new release might render the change(s) unnecessary. Not all issues are safety-critical or mission-critical in a system. Also, it is important to avoid the situation where a change might be more upsetting to users, long-term, than living with the known problem(s) (where the "cure would be worse than the disease"). Basing decisions of the acceptability of some anomalies can avoid a culture of a "zero-defects" mandate, where people might be tempted to deny the existence of problems so that the result would appear as zero *defects*. Considering the collateral issues, such as the cost-versus-benefit impact assessment, then broader debugging techniques will expand to determine the frequency of anomalies (how often the same "bugs" occur) to help assess their impact to the overall system.

Tools

Debugging on video game consoles is usually done with special hardware such as this Xbox debug unit intended for developers.

Debugging ranges in complexity from fixing simple errors to performing lengthy and tiresome tasks of data collection, analysis, and scheduling updates. The debugging skill of the programmer can be a major factor in the ability to debug a problem, but the difficulty of software debugging varies greatly with the complexity of the system, and also depends, to some extent, on the programming language(s) used and the available tools, such as *debuggers*. Debuggers are software tools which enable the programmer to monitor the execution of a program, stop it, restart it, set breakpoints, and change values in memory. The term *debugger* can also refer to the person who is doing the debugging.

Generally, high-level programming languages, such as Java, make debugging easier, because they have features such as exception handling and type checking that make real sources of erratic behaviour easier to spot. In programming languages such as C or assembly, bugs may cause silent problems such as memory corruption, and it is often difficult to see where the initial problem happened. In those cases, memory debugger tools may be needed.

In certain situations, general purpose software tools that are language specific in nature can be very useful. These take the form of *static code analysis tools*. These tools look for a very specific set of known problems, some common and some rare, within the source code, concentrating more on the semantics (e.g. data flow) rather than the syntax, as compilers and interpreters do.

Both commercial and free tools exist for various languages; some claim to be able to detect hundreds of different problems. These tools can be extremely useful when checking very large source trees, where it is impractical to do code walk-throughs. A typical example of a problem detected would be a variable dereference that occurs *before* the variable is assigned a value. As another example, some such tools perform strong type checking when the language does not require it. Thus, they are better at locating likely errors in code that is syntactically correct. But these tools have a reputation of false positives, where correct code is flagged as dubious. The old Unix *lint* program is an early example.

For debugging electronic hardware (e.g., computer hardware) as well as low-level software (e.g., BIOSes, device drivers) and firmware, instruments such as oscilloscopes, logic analyzers, or in-circuit emulators (ICEs) are often used, alone or in combination. An ICE may perform many of the typical software debugger's tasks on low-level software and firmware.

Debugging process

Normally the first step in debugging is to attempt to reproduce the problem. This can be a non-trivial task, for example as with parallel processes and some Heisenbugs. Also, specific user environment and usage history can make it difficult to reproduce the problem.

After the bug is reproduced, the input of the program may need to be simplified to make it easier to debug. For example, a bug in a compiler can make it crash when parsing some large source file. However, after simplification of the test case, only few lines from the original source file can be sufficient to reproduce the same crash. Such simplification can be made manually, using a divide-and-conquer approach. The programmer will try to remove some parts of original test case and check if the problem still exists. When debugging the problem in a GUI, the programmer can try to skip some user interaction from the original problem description and check if remaining actions are sufficient for bugs to appear.

After the test case is sufficiently simplified, a programmer can use a debugger tool to examine program states (values of variables, plus the call stack) and track down the origin of the problem(s). Alternatively, tracing can be used. In simple cases, tracing is just a few print statements, which output the values of variables at certain points of program execution.

Techniques

- *Interactive debugging* uses debugger tools which allow an application's code execution to be processed one step at a time and to be paused to inspect or alter application state. These tools commonly support watchpoints, where execution can proceed until a particular variable changes, and catchpoints which cause the debugger to stop for certain kinds of program events, such as exceptions or the loading of a shared library.
- *Print debugging* or *tracing* is the act of watching (live or recorded) trace statements, or print statements, that indicate the flow of execution of a process and the data progression. Tracing can be done with specialized tools (like with GDB's trace) or by insertion of trace statements into the source code. The latter is sometimes called *printf debugging*, due to the use of the printf function in C. This kind of debugging was turned on by the command TRON in the original versions of the novice-oriented BASIC programming language. TRON stood for, "Trace On." TRON caused the line numbers of each BASIC command line to print as the program ran.
- *Remote debugging* is the process of debugging a program running on a system different from the debugger. To start remote debugging, a debugger connects to a remote system over a communications link such as a local area network. The debugger can then control the execution of the program on the remote system and retrieve information about its state.
- *Post-mortem debugging* is debugging of the program after it has already crashed. Related techniques often include various tracing techniques like examining log files, outputting a call stack on crash,[9] and analysis of memory dump (or core dump) of the crashed process. The dump of the process could be obtained automatically by the system (for example, when the process has terminated due to an unhandled exception), or by a programmer-inserted instruction, or manually by the interactive user.
- *"Wolf fence" algorithm:* Edward Gauss described this simple but very useful and now famous algorithm in a 1982 article for Communications of the ACM as follows: "There's one wolf in Alaska; how do you find it? First build a fence down the middle of the state, wait for the wolf to howl, determine which side of the fence it is on. Repeat process on that side only, until you get to the point where you can see the wolf."[10] This is implemented e.g. in the Git version control system as the command *git bisect*, which uses the above algorithm to determine which commit introduced a particular bug.
- *Record and replay debugging* is the technique of creating a program execution recording (e.g. using Mozilla's free rr debugging tool; enabling reversible debugging/execution), which can be

replayed and interactively debugged. Useful for remote debugging and debugging intermittent, non-determinstic, and other hard-to-reproduce defects.

- *Time travel debugging* is the process of stepping back in time through source code (e.g. using Undo LiveRecorder) to understand what is happening during execution of a computer program; to allow users to interact with the program; to change the history if desired and to watch how the program responds.
- *Delta Debugging* – a technique of automating test case simplification.[11]:p.123
- *Saff Squeeze* – a technique of isolating failure within the test using progressive inlining of parts of the failing test.[12][13]
- *Causality tracking*: There are techniques to track the cause effect chains in the computation.[14] Those techniques can be tailored for specific bugs, such as null pointer dereferences.[15][16]

Debugging for embedded systems

In contrast to the general purpose computer software design environment, a primary characteristic of embedded environments is the sheer number of different platforms available to the developers (CPU architectures, vendors, operating systems, and their variants). Embedded systems are, by definition, not general-purpose designs: they are typically developed for a single task (or small range of tasks), and the platform is chosen specifically to optimize that application. Not only does this fact make life tough for embedded system developers, it also makes debugging and testing of these systems harder as well, since different debugging tools are needed for different platforms.

Despite the challenge of heterogeneity mentioned above, some debuggers have been developed commercially as well as research prototypes. Examples of commercial solutions come from Green Hills Software,[17] Lauterbach GmbH[18] and Microchip's MPLAB-ICD (for in-circuit debugger). Two examples of research prototype tools are Aveksha[19] and Flocklab.[20] They all leverage a functionality available on low-cost embedded processors, an On-Chip Debug Module (OCDM), whose signals are exposed through a standard JTAG interface. They are benchmarked based on how much change to the application is needed and the rate of events that they can keep up with.

In addition to the typical task of identifying bugs in the system, embedded system debugging also seeks to collect information about the operating states of the system that may then be used to analyze the system: to find ways to boost its performance or to optimize other important characteristics (e.g. energy consumption, reliability, real-time response, etc.).

Anti-debugging

Anti-debugging is "the implementation of one or more techniques within computer code that hinders attempts at reverse engineering or debugging a target process".[21] It is actively used by recognized publishers in copy-protection schemas, but is also used by malware to complicate its detection and elimination.[22] Techniques used in anti-debugging include:

- API-based: check for the existence of a debugger using system information
- Exception-based: check to see if exceptions are interfered with
- Process and thread blocks: check whether process and thread blocks have been manipulated
- Modified code: check for code modifications made by a debugger handling software breakpoints
- Hardware- and register-based: check for hardware breakpoints and CPU registers
- Timing and latency: check the time taken for the execution of instructions
- Detecting and penalizing debugger[22]

An early example of anti-debugging existed in early versions of Microsoft Word which, if a debugger was detected, produced a message that said, "The tree of evil bears bitter fruit. Now trashing program disk.", after which it caused the floppy disk drive to emit alarming noises with the intent of scaring the user away from attempting it again.[23][24]

See also

- Assertion (software development)
- Automatic bug fixing
- Debugging pattern
- Magic debug values
- Shotgun debugging
- Software bug
- Software testing
- Time travel debugging
- Trace table
- Troubleshooting

References

1. *"InfoWorld Oct 5, 1981" (https://books.google.com/books?id=JT0EAAAAMBAJ&pg=RA1-PA33) . 5 October 1981. Archived (https://web.archive.org/web/20190918012636/https://books.google.com/boo*

ks?id=JT0EAAAAMBAJ&pg=RA1-PA33&lpg=RA1-PA33&focus=viewport) from the original on September 18, 2019. Retrieved July 17, 2019.

2. "Archived copy" (https://bancroft.berkeley.edu/Exhibits/physics/images/bigscience25.jpg) . Archived (https://web.archive.org/web/20191121001830/https://bancroft.berkeley.edu/Exhibits/physics/images/bigscience25.jpg) from the original on 2019-11-21. Retrieved 2019-12-17.
3. S. Gill, The Diagnosis of Mistakes in Programmes on the EDSAC (https://www.jstor.org/stable/98663) Archived (https://web.archive.org/web/20200306083748/https://www.jstor.org/stable/98663) 2020-03-06 at the Wayback Machine, Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences, Vol. 206, No. 1087 (May 22, 1951), pp. 538-554
4. Robert V. D. Campbell, Evolution of automatic computation (https://dl.acm.org/citation.cfm?id=609784.609786) Archived (https://web.archive.org/web/20190918012641/https://dl.acm.org/citation.cfm?id=609784.609786) 2019-09-18 at the Wayback Machine, Proceedings of the 1952 ACM national meeting (Pittsburgh), p 29-32, 1952.
5. Alex Orden, Solution of systems of linear inequalities on a digital computer (https://dl.acm.org/citation.cfm?id=609784.609793) , Proceedings of the 1952 ACM national meeting (Pittsburgh), p. 91-95, 1952.
6. Howard B. Demuth, John B. Jackson, Edmund Klein, N. Metropolis, Walter Orvedahl, James H. Richardson, MANIAC (https://dl.acm.org/citation.cfm?id=808982) doi=10.1145/800259.808982, Proceedings of the 1952 ACM national meeting (Toronto), p. 13-16
7. The Compatible Time-Sharing System (http://www.bitsavers.org/pdf/mit/ctss/CTSS_ProgrammersGuide.pdf) Archived (https://web.archive.org/web/20120527174321/http://www.bitsavers.org/pdf/mit/ctss/CTSS_ProgrammersGuide.pdf) 2012-05-27 at the Wayback Machine, M.I.T. Press, 1963
8. Peggy Aldrich Kidwell, Stalking the Elusive Computer Bug (https://ieeexplore.ieee.org/document/728224?tp=&arnumber=728224&isnumber=15706) , IEEE Annals of the History of Computing, 1998.
9. "Postmortem Debugging" (https://www.drdobbs.com/tools/postmortem-debugging/185300443) . Archived (https://web.archive.org/web/20191217045909/https://www.drdobbs.com/tools/postmortem-debugging/185300443) from the original on 2019-12-17. Retrieved 2019-12-17.
10. E. J. Gauss (1982). "Pracniques: The 'Wolf Fence' Algorithm for Debugging" (https://dl.acm.org/citation.cfm?id=358690.358695) . Communications of the ACM. **25** (11): 780. doi:10.1145/358690.358695 (https://doi.org/10.1145%2F358690.358695) . S2CID 672811 (https://api.semanticscholar.org/CorpusID:672811) .
11. Zeller, Andreas (2005). Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann. ISBN 1-55860-866-4.
12. "Kent Beck, Hit 'em High, Hit 'em Low: Regression Testing and the Saff Squeeze" (https://web.archive.org/web/20120311131729/http://www.threeriversinstitute.org/HitEmHighHitEmLow.html) . Archived from the original (http://www.threeriversinstitute.org/HitEmHighHitEmLow.html) on 2012-03-11.

13. Rainsberger, J.B. "The Saff Squeeze" (https://blog.thecodewhisperer.com/permalink/the-saff-squeeze) . The Code Whisperer. Retrieved 28 March 2022.

14. Zeller, Andreas (2002-11-01). "Isolating cause-effect chains from computer programs". ACM SIGSOFT Software Engineering Notes. **27** (6): 1–10. doi:10.1145/605466.605468 (https://doi.org/10.1145%2F605466.605468) . ISSN 0163-5948 (https://www.worldcat.org/issn/0163-5948) . S2CID 12098165 (https://api.semanticscholar.org/CorpusID:12098165) .

15. Bond, Michael D.; Nethercote, Nicholas; Kent, Stephen W.; Guyer, Samuel Z.; McKinley, Kathryn S. (2007). "Tracking bad apples". Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications - OOPSLA '07. p. 405. doi:10.1145/1297027.1297057 (https://doi.org/10.1145%2F1297027.1297057) . ISBN 9781595937865. S2CID 2832749 (https://api.semanticscholar.org/CorpusID:2832749) .

16. Cornu, Benoit; Barr, Earl T.; Seinturier, Lionel; Monperrus, Martin (2016). "Casper: Automatic tracking of null dereferences to inception with causality traces" (https://hal.archives-ouvertes.fr/hal-01354090/document) . Journal of Systems and Software. **122**: 52–62. arXiv:1502.02004 (https://arxiv.org/abs/1502.02004) . doi:10.1016/j.jss.2016.08.062 (https://doi.org/10.1016%2Fj.jss.2016.08.062) . ISSN 0164-1212 (https://www.worldcat.org/issn/0164-1212) . S2CID 28161871 (https://api.semanticscholar.org/CorpusID:28161871) . Archived (https://web.archive.org/web/20220125072932/https://hal.archives-ouvertes.fr/hal-01354090/document) from the original on 2022-01-25. Retrieved 2019-06-23.

17. "SuperTrace Probe hardware debugger" (https://www.ghs.com/products/supertraceprobe.html) . www.ghs.com. Archived (https://web.archive.org/web/20171201031136/https://www.ghs.com/products/supertraceprobe.html) from the original on 2017-12-01. Retrieved 2017-11-25.

18. "Debugger and real-time trace tools" (https://www.lauterbach.com) . www.lauterbach.com. Archived (https://web.archive.org/web/20220125072945/https://www.lauterbach.com/frames.html?home.html) from the original on 2022-01-25. Retrieved 2020-06-05.

19. Tancreti, Matthew; Hossain, Mohammad Sajjad; Bagchi, Saurabh; Raghunathan, Vijay (2011). "Aveksha: A Hardware-software Approach for Non-intrusive Tracing and Profiling of Wireless Embedded Systems". Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems. SenSys '11. New York, NY, USA: ACM: 288–301. doi:10.1145/2070942.2070972 (https://doi.org/10.1145%2F2070942.2070972) . ISBN 9781450307185. S2CID 14769602 (https://api.semanticscholar.org/CorpusID:14769602) .

20. Lim, Roman; Ferrari, Federico; Zimmerling, Marco; Walser, Christoph; Sommer, Philipp; Beutel, Jan (2013). "FlockLab: A Testbed for Distributed, Synchronized Tracing and Profiling of Wireless Embedded Systems". Proceedings of the 12th International Conference on Information Processing in Sensor Networks. IPSN '13. New York, NY, USA: ACM: 153–166. doi:10.1145/2461381.2461402 (https://doi.org/10.1145%2F2461381.2461402) . ISBN 9781450319591. S2CID 447045 (https://api.semanticscholar.org/CorpusID:447045) .

21. *Shields, Tyler (2008-12-02). "Anti-Debugging Series – Part I" (https://www.veracode.com/blog/2008/12/anti-debugging-series-part-i) . Veracode. Archived (https://web.archive.org/web/20161019075351/http://www.veracode.com/blog/2008/12/anti-debugging-series-part-i) from the original on 2016-10-19. Retrieved 2009-03-17.*

22. *"Software Protection through Anti-Debugging Michael N Gagnon, Stephen Taylor, Anup Ghosh" (https://web.archive.org/web/20111001172453/http://people.seas.harvard.edu/~mgagnon/software_protection_through_anti_debugging.pdf) (PDF). Archived from the original (http://people.seas.harvard.edu/~mgagnon/software_protection_through_anti_debugging.pdf) (PDF) on 2011-10-01. Retrieved 2010-10-25.*

23. *Ross J. Anderson (2001-03-23). Security Engineering (https://archive.org/details/securityengineer00ande) . p. 684. ISBN 0-471-38922-6.*

24. *"Microsoft Word for DOS 1.15" (http://toastytech.com/guis/word1153.html) . Archived (https://web.archive.org/web/20130514033750/http://toastytech.com/guis/word1153.html) from the original on 2013-05-14. Retrieved 2013-06-22.*

Further reading

- Agans, David J. (2002). *Debugging: The Nine Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems* (https://archive.org/details/debugging9indisp0000agan) . AMACOM. ISBN 0-8144-7168-4.
- Blunden, Bill (2003). *Software Exorcism: A Handbook for Debugging and Optimizing Legacy Code*. APress. ISBN 1-59059-234-4.
- Ford, Ann R.; Teorey, Toby J. (2002). *Practical Debugging in C++* (https://archive.org/details/practicaldebuggi00ford) . Prentice Hall. ISBN 0-13-065394-2.
- Grötker, Thorsten; Holtmann, Ulrich; Keding, Holger; Wloka, Markus (2012). *The Developer's Guide to Debugging, Second Edition*. Createspace. ISBN 978-1-4701-8552-7.
- Metzger, Robert C. (2003). *Debugging by Thinking: A Multidisciplinary Approach*. Digital Press. ISBN 1-55558-307-5.
- Myers, Glenford J (2004). *The Art of Software Testing* (https://archive.org/details/artofsoftwaretes00myer) . John Wiley & Sons Inc. ISBN 0-471-04328-1.
- Robbins, John (2000). *Debugging Applications*. Microsoft Press. ISBN 0-7356-0886-5.
- Telles, Matthew A.; Hsieh, Yuan (2001). *The Science of Debugging*. The Coriolis Group. ISBN 1-57610-917-8.
- Vostokov, Dmitry (2008). *Memory Dump Analysis Anthology Volume 1*. OpenTask. ISBN 978-0-9558328-0-2.

- Zeller, Andreas (2009). *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*. Morgan Kaufmann. ISBN 978-0-1237-4515-6.

External links

Wikiquote has quotations related to ***Debugging***.

The Wikibook *Computer Programming Principles* has a page on the topic of: ***Debugging***

- Crash dump analysis patterns (http://www.dumpanalysis.org/) – in-depth articles on analyzing and finding bugs in crash dumps
- Learn the essentials of debugging (https://web.archive.org/web/20070218145734/http://www-128.ibm.com/developerworks/web/library/wa-debug.html?ca=dgr-lnxw03Dbug) – how to improve your debugging skills, a good article at IBM developerWorks (archived from the original on February 18, 2007)
- Plug-in Based Debugging For Embedded Systems (http://www.clarinox.com/docs/whitepapers/EmbeddedDebugger.pdf)
- Embedded Systems test and debug – about digital input generation (https://web.archive.org/web/20120112200659/http://www.byteparadigm.com/embedded-systems-test-and-debug---about-digital-input-generation-135.html) – results of a survey about embedded system test and debug, Byte Paradigm (archived from the original on January 12, 2012)

Wikipedia