Introduction to Debugging

Important Notice : Courseware - Legal

This courseware is both the product of the author and of freely available opensource and/or public domain materials. Wherever external material has been shown, it's source and ownership have been clearly attributed. We acknowledge all copyrights and trademarks of the respective owners.

The contents of the **courseware PDFs are considered proprietary** and thus cannot be copied or reproduced in any form whatsoever without the explicit written consent of the author.

Only the programs - **source code** and binaries (where applicable) - that form part of this courseware, and that are made available to the participant, are released under the terms of the permissive **MIT license** [1].
Under the terms of the MIT License, you can certainly use the source code provided here; you must just attribute the original source (author of this courseware and/or other copyright/trademark holders).

VERY IMPORTANT :: Before using this source(s) in your project(s), you *MUST* check with your organization's legal staff that it is appropriate to do so.

The courseware PDFs are *not* under the MIT License, they are to be kept confidential, non-distributable without consent, for your private internal use only.

The duration, contents, content matter, programs, etc. contained in this courseware and companion participant VM are subject to change at any point in time without prior notice to individual participants.

Care has been taken in the preparation of this material, but there is no warranty, expressed or implied of any kind, and we can assume no responsibility for any errors or omisions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.


DEBUGGING
/DE-BUHG-ING/ VERB
1. Being the detective in a crime movie where you are also the murderer.

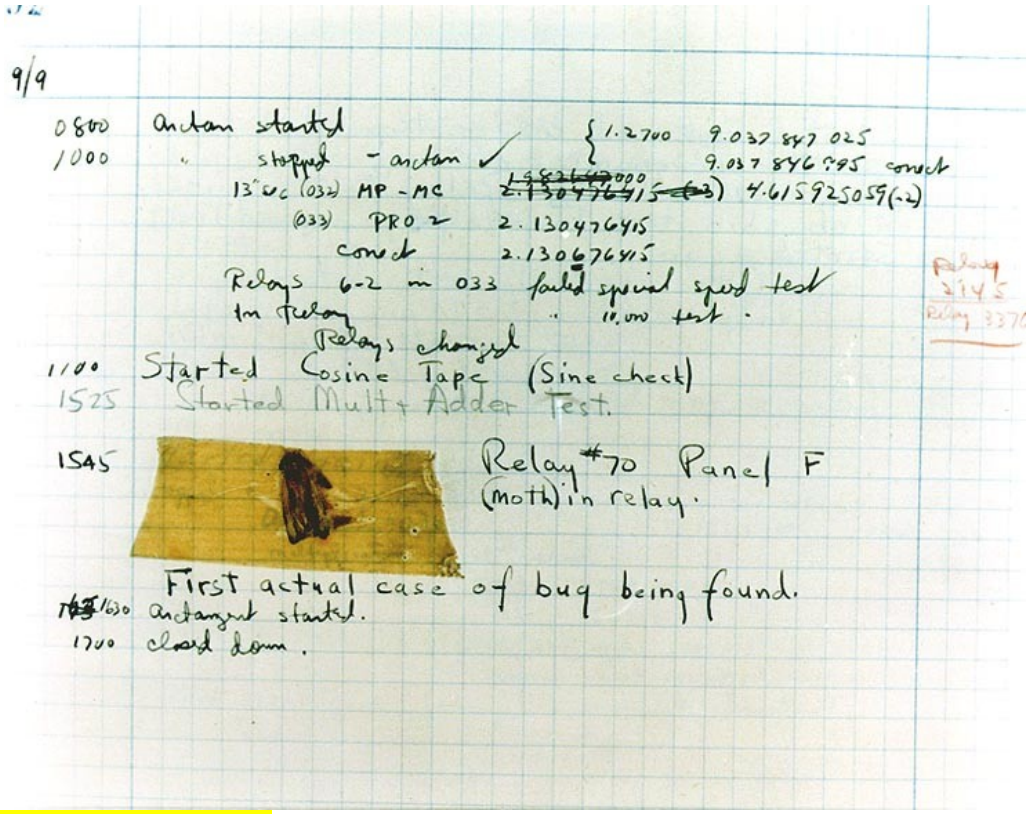
Src

See:

- Start with "Software Horror Stories"

- YouTube video: Engineering Disasters 13 - Software Flaws (Patriot Missile System failure)

- "Debugging" on Wikipedia
  URL: http://en.wikipedia.org/wiki/Debugging



Description H96566k.jpg

The First "Computer Bug" Moth found trapped between points at Relay # 70, Panel F, of the Mark II Aiken Relay Calculator while it was being tested at Harvard University, 9 September 1945. The operators affixed the moth to the computer log, with the entry: "First actual case of bug being found". They put out the word that they had "debugged"

the machine, thus introducing the term "debugging a computer program". In 1988, the log, with the moth still taped by the entry, was in the Naval Surface Warfare Center Computer Museum at Dahlgren, Virginia.

Removed caption read: Photo # NH 96566-KB First Computer "Bug", 1945

Source: U.S. Naval Historical Center Online Library Photograph NH 96566-KN

Date: September 1945

Author: Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988.

- *Source: "The Art of Debugging" by Matloff & Salzman, No Starch Press.*
  The Essence of Debugging is the Principle of Confirmation.
  Fixing a buggy program is a process of confirming, one by one, that the many things you *believe* to be true about the code actually *are* true. When you find that one of your assumptions is *not* true, you have found a clue to the location (if not the exact nature) of a bug.

- "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."
  Brian Kernighan.

- "Simplicity is the ultimate sophistication."

Leonardo da Vinci.

- "The price of reliability is the pursuit of the utmost simplicity."
  C.A.R. Hoare

- Testing can, at best, confirm the presence of errors, not their absence.

Another really interesting true story - Design by Contract: The Lessons of Ariane:
how software assertions could have saved half-a-billion dollars! #Ariane #debugging #assertions

Please read this blog post on what exactly an **assertion** is and how they are to be used and how they are *not* to be used.

The Art of Debugging : https://github.com/stas00/the-art-of-debugging
Fast Debugging Methodology : https://github.com/stas00/the-art-of-debugging/blob/master/methodology

Risks Forum

http://catless.ncl.ac.uk/Risks/22.13.html

Software problem kills soldiers in training incident
[http://catless.ncl.ac.uk/Risks/22.13.html#subj2.1 **]**

Steve Bellovin <smb@research.att.com>
Thu, 13 Jun 2002 09:38:10 -0400

According to a U.S. Army report, a software problem contributed to the deaths of two soldiers in a training accident at Fort Drum. They were firing artillery shells, and were relying on the output of the Advanced Field Artillery Tactical Data System. But if you forget to enter the target's altitude, the system assumes a default of 0. (A Web site I found indicates that (part of) Ft. Drum is at 679 feet above sea level.) The report goes on to warn that soldiers should not depend exclusively on this one system, and should use other computers or manual calculations.

Other factors in the incident include the state of training of some of the personnel doing the firing. [Source: AP]

Steve Bellovin, http://www.research.att.com/~smb (me)
http://www.wilyhacker.com ("Firewalls" book)

Confirming cricket score reason for delay

"*R. Jagannathan*" *<jagan@reactivenetwork.com>*
Tue, 4 Jun 2002 00:48:23 -0700

http://www.cricket.org/link_to_database/ARCHIVE/CRICKET_NEWS/2002/JUN/012194_NATION_03JUN2002.html

An interesting article from a "dependability" standpoint. Mismatch between what a computer thought was the revised target and what the umpire manually computed caused a 20-minute delay during which the teams had to play without knowing the revised target. The game between India and West Indies was won by India and the West Indian captain later rued the fact that their team did not know the score for 20 minutes, which affected how they played. All this for a one-run difference between the manual and computed calculation.

The Duckworth-Lewis method is an empirically-derived table (by the two mathematicians) to compute revised targets when a one-day cricket match gets shortened by rain or other similar disruptions.

This #Car Runs on #Code

BY Robert N. Charette // February 2009

“The avionics system in the F-22 Raptor, the current U.S. Air Force frontline jet fighter, consists of about 1.7 million lines of software code. The F-35 Joint Strike Fighter, scheduled to become operational in 2010, will require about 5.7 million lines of code to operate its onboard systems. And Boeing’s new 787 Dreamliner, scheduled to be delivered to customers in 2010, requires about 6.5 million lines of software code to operate its avionics and onboard support systems.

These are impressive amounts of software, yet if you bought a premium-class automobile recently, ”it probably contains close to 100 million lines of software code,” says Manfred Broy, a professor of informatics at Technical University, Munich, and a leading expert on software in cars. All that software executes on 70 to 100 microprocessor-based electronic control units (ECUs) networked throughout the body of your car.

…”

Read the full article here:
http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code

#embedded #software

[Article snippets below]
*Source: “**The ten secrets of embedded debugging**”*
By Stan Schneider and Lori Fraleigh, Embedded.com
Sep 15 2004 (16:30 PM)
URL: http://www.embedded.com/showArticle.jhtml?articleID=47208538

“Debugging your system may be the most important step in the development process. Here are ten hard-won lessons from the embedded trenches.”

--snip--

Pursue quality to save time

Studies show that 74% of statistics are made up. More seriously, our

experience indicates that over 80% of development time is spent:
Debugging your own code
Debugging the integration of your code with other code
Debugging the overall system

Worse, it costs 10 to 200 times more to fix a bug at the end of the cycle than at the beginning. The cost of a small bug that makes it to the field can be astronomical. Even if the bug doesn't have a significant impact on performance, it can significantly affect perceived quality. Just ask the developers of the original Pentium's floating-point microcode. Several years ago, Intel shipped millions of chips with an insignificant flaw. Even though the flaw affected almost no applications, it was a public-relations disaster for Intel, caused its first chip recall, and forced an eventual charge against company earnings of $475 million.

You can't achieve quality by saving testing until the end of development. A quality process, such as testing as you go, reduces the critical "what's going on" time and greatly speeds development. Get into the habit of testing and scanning for problems every day. You'll ship better code faster.

---

http://www.ganssle.com/item/automatically-debugging-firmware.htm – On DBC – Design By Contract, as Eiffel does

"... Software engineering is topsy-turvey. In all other engineering disciplines it's easy to add something to increase operating margins. Beef up a strut to take unanticipated loads or add a weak link to control and safely manage failure. Use a larger resistor to handle more watts or a fuse to keep a surge from destroying the system. But in software a single bit error can cause the system to completely fail. DBC is like adding fuses to the code. The system is going to crash; it's better to do so early and seed debugging breadcrumbs than fail in a manner that no one can subsequently understand. ..."

Source: "The Embedded Muse" Newsletter (# 172 dt 6 Jan 2009), edited by Jack Ganssle.

--snip--

Response to Last Issue's Quotes and Thoughts

> We know about as much about software quality problems as they knew
> about the Black Plague in the 1600s. We've seen the victims'
agonies

> and helped burn the corpses. We don't know what causes it; we don't
> really know if there is only one disease. We just suffer - and keep
> pouring our sewage into our water supply.

Steve Litt disagreed: I disagree with this entirely. On one level, ***we know exactly what causes bugs:***

* Uninitialized variables and pointers

* Overrunning arrays

* Aiming pointers at the wrong place

* Using local strings as function returns

* Picket fence conditions

* Etcetera

Perhaps he meant we don't know how it could be solved. Again, I at least partially disagree. If programmers had the time and compensation to code without bugs, they could:

* Code modularly

* Document each subroutine: input, output and intentional side effects

* Use proper and consistent naming and indentation

* No hotdog coding. One-liner loops aren't a badge of coolness.

* Build a test jig for each subroutine and test under varying conditions

* Test like that at each level up the tree, up and up to the top

* gcc -Wall, then fix every single warning

* Get lint and use it

* Get and use valgrind or another leak finder

* Get and use a memory tromp finder (maybe run it under VMS :-)

3RD PARTY LIBRARY DEFENSIVE CODING

* If it isn't exquisitely documented, don't use it.

* If its source isn't available, be very afraid

* But don't change its source unless absolutely necessary

* Run valgrind or another leak finder on it first, dump it if it leaks or doesn't give back at the end and the vendor can't give a good explanation

* Run a memory tromp finder, dump it there are memory tromps the vendor can't quickly fix or give a good explanation for.

* Build several test jigs for it, and put it through its paces. Dump it if the vendor can't explain and ameliorate every failure.

Of course, we don't have time for this. Time to market is king. The company who actually did this might get scooped every time and go out of business, or at least that's the prevailing thought. The coder who does this will be slower (according to prevailing thought) and will get fired.

Meanwhile, the customers prioritize newer, cheaper, sooner over bug free. It's not much different from people enduring the hassle of shopping at Wal-Mart because they save money.

So bottom line, the human cause of bugs is nobody is willing to pay to keep them out.
--snip--

Credit: Julia Evans (below)

The Embedded Muse 374 (May 2019)

Plane Crashes and Feedback Loops

Photos of cats, celebrities, and disasters flood the internet. But to me, this image is one of the most compelling I've seen:


debugging: ♥ love your bugs ♥
(thanks to Allison Kaptur for teaching me this attitude! she has a great talk called "Love Your Bugs".)
Debugging is a great way to learn. First, the harsh reality of bugs in your code is a good way to reveal problems with your mental model.
program
error: too many open files
I can't just open as many files as I want? Interesting!
Fixing bugs is also a good way to learn to write more reliable code!
hmm, I should put in error handling here in case that database query times out.
Also, you get to solve a mystery and get immediate feedback about whether you were right or not.
that's weird...
oh goodness, that's a lot of errors
I have an idea!
coding a fix
it works now!
great work!
Nobody writes great code without writing + fixing lots of bugs. So let's talk about debugging skills a bit!
JULIA EVANS

nearly three orders of magnitude better than most good-quality firmware.

We all make mistakes and bugs will always be an element of creating software, which is probably the most complex engineered product in history. But great teams will figure out *how* errors creep in and patch the process. This feedback loop helps teams evolve to a higher level of perfection.

Source: The Embedded Muse 234

A Shuttle Failure Averted

The second Space Shuttle launch in 1981 was delayed by a month when a fuel spill loosened a number of its tiles. The crew booked simulator time in Houston to practice a number of scenarios, including a "Transatlantic Abort", where the vehicle can neither make orbit nor can return to the launch site. Suddenly, all four main Shuttle computers (in the simulator) crashed. Part of the abort scenario requires fuel dumps to lighten the spacecraft. It was during the second of these dumps that the crash occurred.

After a couple of days of analysis, programmers discovered that the fuel management module, which had done one dump and successfully exited, when recalled for the second fuel dump, restarted thinking that this was it's first incarnation. Counters in the code had not been properly reinitialized, though, causing what was essentially a computed GOTO to branch out of the code, into a random section of memory.

A simple fix took care of the problem. More interesting, though, was the realization that this same sort of bug had appeared in different modules before. The programmers were committed to producing only the best code, so decided to see if they could come up with a systematic way to eliminate these generic sorts of bugs in the future.

They developed a list of seven questions that had a high probability of isolating similar problems. A different group of programmers then applied these questions to the bad fuel dumping module (to see if they would indeed pick up the known problem), as well as other modules.
The questions detected every one of the bugs. 17 additional, previously unknown, problems surfaced!

This is software engineering at its best. Instead of quickly fixing the bug and moving on - as most of us do - the development team transformed the defect into an opportunity to improve the code.

Random :-)

Quotes and Thoughts

Always code as if the guy who ends of up maintaining your code will be a violent psychopath who knows where you live. Damian Conway.
Source: the Embedded Muse 226, Jack Ganssle

Thoughts-

We want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only secondarily for machines to execute. - Harold Abelson and Gerald Jay Sussman.

Src: The Embedded Muse

Jokes About Computers and Embedded Systems

```
The World's Last C Bug:

 while (1)
 {
   status = GetRadarInfo();
   if (status = 1)
     LaunchMissiles();
 }
```
