Techniques for Debugging

Interrogating unfamiliar code

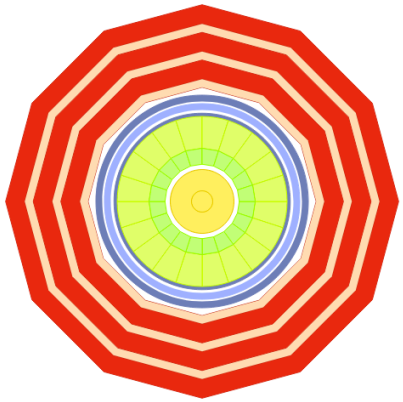
Andy Boughton

Abecasis Group Meeting

February 25, 2020

Demonstration: Cheating at Particle Clicker

- A JavaScript game
  - Not a very good game
  - Designed to waste many hours of your life, just... clicking
  - Basically, it's like most web games
- We will defeat it in minutes



https://particle-clicker.web.cern.ch/particle-clicker/

Legal Disclaimer:

Cheating is bad. Don’t do it.

General Strategies

First: Take an inventory

Identify what is expected

- What is this program intended to do? (Features, audience)
- Simplify: What is the smallest reproducible example?
- Do you know what inputs/ outputs are expected?
- Check “ground truth”: does the program actually work?
  - Unit tests, tutorials, etc

Plan how to interrogate the system

- Relate features to code: what do you need in order to test a hypothesis?
- Can you locate a copy of the code?
  - Is it the same as the version of the program you are running?
- Can the program be compiled and run from scratch?
- What configuration options might affect the results?

"Novice programmers showed a "fairly strong linear character" with 70% of their eye movements on source code being linear, compared with 60% for expert programmers. It is suggested this reflects the experts' ability to follow the execution order of a program and/or to seek out beacons in the code as an aid to understanding"

https://link.springer.com/article/10.1007/s10664-018-9649-y

Simplest reproducible test case

- Complexity makes it hard to find the root cause
  - Simplify the problem: Faster to run & less code to look at
- Build a hypothesis for how this happened: translate code to behavior
  - What is the observable outcome? How did it occur?
  - Could we have found it sooner? (validation, sanity checks, etc)
  - How did we get lucky? (is this the worst-case scenario?)
- After you fix the bug, test case becomes basis for automated regression test: verify that the bug stays fixed

https://blog.newrelic.com/technology/blameless-retrospectives/

Proactive vs Reactive

- *print* statements are a simple debugging strategy
  - But they only work for the questions you think to ask (reactive)
  - Very verbose, but maybe not the info you need
  - Capturing more information requires running the program again
- Proactive strategies *plan for common scenarios* and capture relevant information, in a structured way that is easier to search
  - assert statements verify basic assumptions (“n > 0”)
  - Unit tests to verify code in advance
  - Automatic error capturing (eg Sentry)
  - Log files: include normal events, which helps to establish sequence and timeline (eg Apache access and error logs)

The Value of Domain Knowledge

- Design shortcuts can lead to surprises
  - Floating point underflow in numerical calculations (very small p-values)
  - Weird data: Unicode text, Y2K bug / 2038 problem
- Language features can affect what to look for. Examples:
  - Threads or async → Race conditions
  - Memory management → Memory leaks, security issues, etc
  - Dynamic or weak typing → Check that variables are what you expect
- Sometimes, abstractions create new failure modes. Examples:
  - Network drives hang or disappear
  - Two separate processes compete for the same resources
  - Caching. Oh god. Caching.

Tools and Techniques

Read
Eval
Print
Loop

"A read–eval–print loop (REPL), also termed an interactive toplevel or language shell... takes single user inputs (i.e., single expressions), evaluates (executes) them, and returns the result to the user; a program written in a REPL environment is executed piecewise... Common examples include command line shells and similar environments for programming languages, and the technique is very characteristic of scripting languages."

https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop

When to use a REPL?

Strengths

- Great for exploration and building operations step by step
- Explore variables and data structures in great detail
- Manipulate a system to try several ideas without rerunning programs
- Common in dynamic languages (Python, R, JS, PHP, etc)
- Can take many forms (eg Jupyter notebook)

Weaknesses

- Traceability: what is the code that actually produced your result?
  - All your experiments are mixed together
- Not every variable you want to see is accessible (in global scope)
- Does not work well with complex or slow-running code
- May not exist for common compiled languages

Inputs and Outputs

- If the program internals are not accessible, we can learn about it by examining its effects upon the world
  - How does it receive data?
  - How does it store and display results?
  - Are there unit tests describing expected behavior?
- For a complex program, this provides a starting point to understand *execution flow,* rather than reading the code linearly

Monitoring Inputs and Outputs

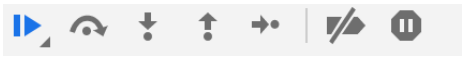
- strace can tell you what files are being used
- Network monitoring tools can tell you about connections to other services (browser DevTools, tcpdump, etc)
- CPU and I/O monitoring tools can identify bottlenecks
- If you are experimenting, track versions carefully
  - Being organized is a big part of debugging

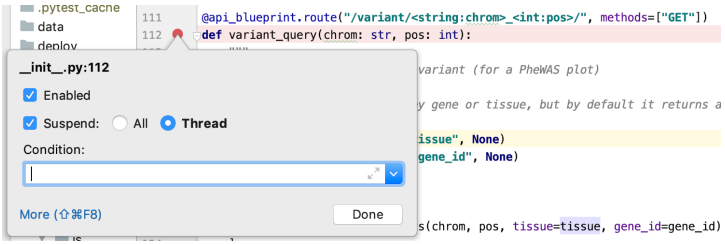
Environment: Understand Your Surroundings

- A program is more than code and data
- The environment it runs in can affect behavior
  - Language version & dependencies
  - Hardware
  - Operating system / package updates
  - Global or user-specific configuration
- In web applications, sometimes the rules change* over time (!)

* https://caniuse.com/
* https://blog.chromium.org/2019/10/developers-get-ready-for-new.html

Breakpoint Debuggers



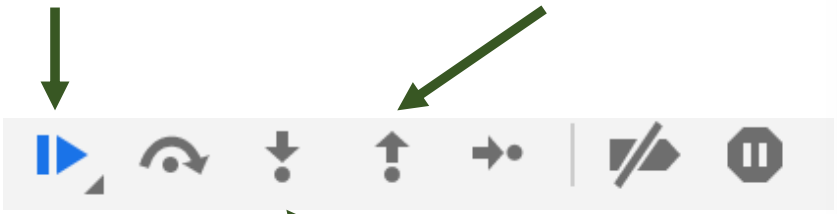


- Very powerful way to pause or follow execution at a specified point
- Watch expressions provide more control over when to pause
- Can inspect or modify variables
- Call stack provides context
- Many languages have these tools, and IDEs integrate nicely

Stepping Through Code with Breakpoints

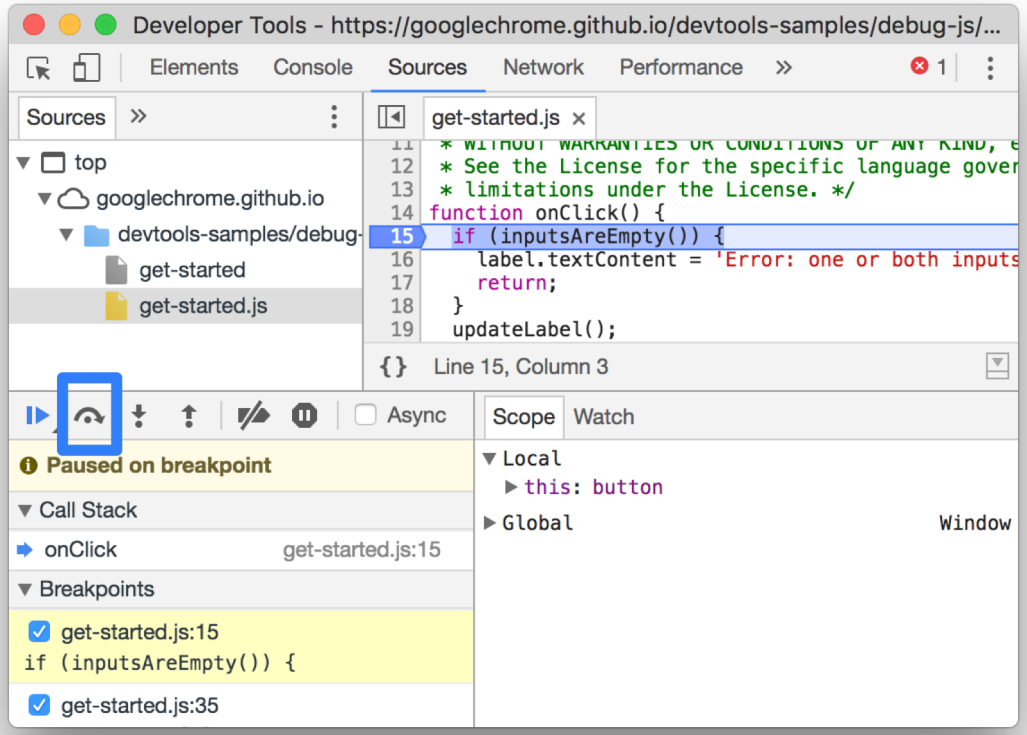
Resume:
Run code as normal

Step out:
Leave a function call



Step into:
Go into a function call

Step over:
Don’t go into a function call



https://developers.google.com/web/tools/chrome-devtools/javascript/reference#stepping

Downsides of Breakpoints

- Too much context to process: in a big problem, there may be **a lot** of code to step through, & functions may get called repeatedly
  - Conditional breakpoints or watch expressions can help
  - Not all the code is yours! (blackbox libraries or step over)
- Using a debugger can incur a major performance penalty: you may wait a long time to hit the breakpoint
- Editing variable contents may change the program: you could introduce bugs by violating assumptions of the code
  - "When you have to debug your debugging, you've gone too far"

Advanced techniques

- Static analysis and other tools: automate error detection
  - Linting, valgrind, purify, type checkers, etc
- Linux tools: strace, lsof, perf trace, etc: monitor specified program (even if already running)
  - Files that get opened (what information is the program using?)
  - Identify dependencies
  - Timing information: where did it get stuck?
- CPU and memory profiling tools: why is a thing slow? (perf, dstat, etc)
- Learn the special tricks for your language and domain
  - “Magic expressions” for Jupyter, DevTools for the web, R “traceback”/”recover”, etc...

Further Reading

- Jon Skeet's Debugging tips: https://jonskeet.uk/csharp/debugging.html
- "The 500 mile email", and other debugging stories: https://github.com/danluu/debugging-stories
- Best practices:
  - Bug reports: https://imagej.net/Bug_reporting_best_practices
  - Logging: https://www.loomsystems.com/blog/single-post/2017/01/26/9-logging-best-practices-based-on-hands-on-experience
- Wizard Zines: approachable intros to useful tools
  - Debugging (overview of many tools): https://jvns.ca/debugging-zine.pdf
  - Profiling & tracing: https://wizardzines.com/zines/perf/
  - strace: https://wizardzines.com/zines/strace/
  - tcpdump: https://wizardzines.com/zines/tcpdump/