Computing at Scale

Lecture 15: Debugging (gdb)

Jacob Merson

March 13, 2025

Debugging

Introduction

So far, we have focused on the syntax and semantics and tooling.

- Did anyone's code work on the first try?
- What strategies did you try to fix the problems?

What is debugging

Debugging is a process by which you find and fix errors in your code.

The term is often attributed to Admiral Grace Hopper from the 1940s when her colleague found a moth in their Mark II computer which prevented operations and she remarked that they were “debugging”. It is also believed that it was originated by others much earlier.

Cost of debugging

Software developers spend 35-50 percent of their time validating and debugging software. The cost of debugging, testing, and verification is estimated to account for 50-75 percent of the total budget of software development projects, amounting to more than $100 billion annually. — O'Dell (2017)

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it? — Kernighan and Plauger (1978)

Types of Errors

- **Syntax errors:** Errors that arise from invalid syntax.
- **Logic errors:** Errors that come from programming invalid logic.
- **Semantic errors:** Errors that occur from missuses of the programming language.
- **Runtime errors:** Errors that occur during the runtime which occur due to the computer system. Examples are out of memory errors or stack overflow.

Debugging Strategies

Overview

Errors arise from a incorrect assumptions about the state of data, variables, or program semantics. Sometimes errors arise from simple typos.

Strategies for finding errors:

- C++ compiler error messages
- `print` debugging
- “Rubber Duck” debugging
- reduce code to minimal example
- debuggers / pdb
- testing / assertions

Hypothesis Testing

Debugging is done in a few steps that you repeat until you find the problem:

1. Form a hypothesis about why your code may not be working
2. Check your hypothesis
3. Implement fix or return to step 1

Print Debugging

With print debugging we use print statements to check our hypothesis and determine the program state.

Rubber Duck Debugging

The idea of rubber duck debugging is to explain your program to a rubber duck. The act of explaining what your code does aloud often helps expose what might be going wrong. It sounds *crazy*, but it works!

Rubber Duck Debugging (continued)

1. *Beg, borrow, steal, buy, fabricate or otherwise obtain a rubber duck (bathtub variety).*
2. *Place rubber duck on desk and inform it you are just going to go over some code with it, if that's all right.*
3. *Explain to the duck what your code is supposed to do, and then go into detail and explain your code line by line.*
4. *At some point you will tell the duck what you are doing next and then realise that that is not in fact what you are actually doing. The duck will sit there serenely, happy in the knowledge that it has helped you on your way.*

***Note**: In a pinch a coworker might be able to substitute for the duck, however, it is often preferred to confide mistakes to the duck instead of your coworker. — https://rubberduckdebugging.com/*

Minimal working example

Often we are debugging large and complex programs. One way to find a bug is to start reducing and simplifying your code until you have a bare bones example of the bug. During this simplification process, you often find the problem. If not, it provides a easy way to get help quickly.

Testing/Assertion

- **Unit tests:** write test cases to check if your functions are behaving as expected with a given set of input data.
- **Assertions:** liberally use assertions to check function pre and post conditions. This will give you a nice stacktrace to work with if something unexpected happens.

Debuggers

Why use a debugger?

Debuggers provide an interactive way to step through your python source code line by line and investigate the program state. Once you gain proficiency with a debugger it is often much faster than `print` debugging.

Common Debuggers:

- gdb
- lldb
- totalview (parallel)
- ddt (parallel)

Running a program in gdb

- To run a program in gdb, you need to compile your program with the `-g` flag to include debugging information. In CMake you can simply use the `CMAKE_BUILD_TYPE` variable to set the build type to `Debug`.
- `gdb ./my_program` or `gdb --args ./my_program arg1 arg2`
- `gdb -p <pid>` attach to a running process

- `run` or `r` to run the program
- `start` to start the program and stop at the first line (breakpoint at `main`)
- `break` or `b` to set a breakpoint
- `print` or `p` to print a variable
- `next` or `n` to step to the next line
- `step` or `s` to step into a function
- `continue` or `c` to continue execution
- `list` or `l` to list the source code
- `quit` or `q` to quit gdb

- `dprintf` dynamic printf (adds printf) without recompiling
- `backtrace` or `bt` to print the stack trace
- `frame` to select the frame
- `watch` to set a watchpoint
- `info` and `info locals` to get information about the program and variables

GDB Extras

- `gdb -tui` to run gdb in text user interface mode
- no need to restart gdb after recompiling, just use `run` again
- Excellent user manual
  `https://sourceware.org/gdb/current/onlinedocs/gdb.html/`

Try running gdb on a simple program

References

- Kernighan, Brian W., and P. J. Plauger. 1978. *The Elements of Programming Style*. 2d ed. New York: McGraw-Hill.
- O'Dell, Devon H. 2017. "The Debugging Mindset: Understanding the Psychology of Learning Strategies Leads to Effective Problem-Solving Skills." *Queue* 15 (1): 71–90. `https://doi.org/10.1145/3055301.3068754`.