INTERMEZZO C: GDB and the Fine Art of Debugging

Gene Cooperman

C GDB and the Fine Art of Debugging

C.1 The Core Conflict: Manifestation vs. Root Cause

In debugging, we deal with two distinct entities:

1. **The Manifestation:** This is the symptom. It's the "Segmentation Fault," the "Assertion Failed" message, or the incorrect value printed at the end of a calculation.
2. **The Root Cause:** This is the actual logic error—the uninitialized variable, the "off-by-one" index, or the race condition—that occurred perhaps thousands of lines of code or millions of CPU cycles before the manifestation appeared.

The **Art of Debugging** is the process of tracing the manifestation backward through time and logic to find the root cause. Your goal is to narrow the gap between the moment the error happens and the moment the program tells you something is wrong.

C.2 The Three Levels of Debugging

C.2.1 Level 1: Print Statements (The Quick Look)

- **Method:** Inserting `printf()` or `cout` to track state.
- **The Limitation:** For long-running programs, this produces a "wall of text." It can also cause **Heisenbugs**, where the act of printing changes the program's timing and hides the bug.

C.2.2 Level 2: Assertions and Defensive Programming

- **Method:** Defining what the state *must* be using `assert()`.
- **Goal:** Force the program to stop **immediately after the root cause occurs**.
- **Defensive Syscalls:** Always check return codes.

  Example:

```
int rc = signal(SIGUSR2, &handler);
if (rc == -1) {
    perror("signal"); // Explains WHY it failed
```

```
        exit(1);
    }
```

C.2.3 Level 3: Using a Good Debugger like GDB

The guide in the next section provides an overview of essential GDB commands and setup instructions for different operating systems. Using even a few of these commands effectively can save hours of manual debugging.

C.3 The Basics of GDB (GNU DeBugger)

C.3.1 Platform Setup

If you are using a current Mac (macOS), **GDB is not available**. You have two options:

- **Virtualization:** Create a Linux virtual machine on your Mac to use GDB.
- **Native Alternative (LLDB):**
  - 1. Install command line tools: `xcode-select --install`.
  - 2. Invoke **lldb** instead of gdb (e.g., `lldb a.out`).
  - 3. Refer to the GDB to LLDB command map for command translations.

C.3.2 Getting Started

- Invoking GDB
  To start GDB with specific program arguments:

  Command: `gdb --args <executable> <args>`

  Example: `gdb --args ./a.out arg1 arg2`

- Starting and Stopping Execution
  - **Set Initial Breakpoint:** `break main`
  - **Launch Program:** `run`
  - **Exit:** `quit`.

C.3.3 Essential Commands

- **Where Am I? (Inspection)**
  - **Threads:** `info threads` (List all threads if this process has more than one thread).
  - **Switch Thread:** `thread 1` (Inspect thread number 1).
  - **Backtrace:** `where` or `backtrace` or `bt` (Displays the call stack).
  - **Frames:** `frame 2` or `f 2` (Move to a specific stack call frame).
  - **Source Code:** `list` or `l` (Displays the code around the current line).
- **Printing & Examining**
  - **Check Type:** `ptype <variable>` (e.g., `ptype argv[0]`).
  - **View Value:** `print <variable>` (e.g., `print argv[0]`).

- **Listing:** Try `list myfunction` or `list myfile:17` for line number 17.
- **Finding function names:** `info function SUBSTRING` for `SUBSTRING` a part of a function name. (Useful for discovering fully qualified names in C++.)

- **Breakpoints**
  - **`break`**: Try `break myfunction` or `break myfile:17` for line number 17.
  - **`info breakpoints`**: List all breakpoints.
  - **`disable`**: Try `disable 2` to disable breakpoint number 2.
  - **`enable`**: Try `enable 2` to re-enable breakpoint number 2.
- **Execution Control**
  - **`next`**: Move to the next line of code.
  - **`step`**: Step inside a function call.
  - **`until`**: Continue until reaching a higher line number (useful to escape a "for loop").
  - **`finish`**: Finish the current function and return to the caller.
  - **`continue`**: Resume execution until the next breakpoint is hit.

(**HINT:** Most of these commands can take a numeric argument for executing multiple times.)

C.3.4 Advanced Debugging & Tips

- **Special Scenarios**
  - **Infinite Loops:**
    - Execute `run` in GDB, and then type **control-c** (`^C`) to interrupt the program and talk to GDB. Once interrupted, use `where` or `print` to find the loop's location.
  - **Multiprocessing:** Use `set follow-fork-mode child` to switch debugging to the child process after a `fork()` call. The default is to continue to debug the parent process.
- **Productivity**
  - **Auto-complete:** Use the **TAB** key.
  - **Navigation: Cursor keys** work for command history.
  - **Documentation:** Use `help <command>` (e.g., `help continue`).
  - **Browsing versus Command Mode:** Switch back and forth between a mode for browsing the source code, and the normal command mode.
    (**NOTE:** `^X` is the notation for `control-x`, i.e., pressing the control key, and then typing `x`.)
    - To switch to source code browsing mode, do any of:
      `tui enable` or `layout src` or simply `^Xa`
      The cursor keys will then be used to browse the source code.
    - Do ^Xa again to switch back to command mode.
    - While in browsing mode, try `focus cmd` or `focus src` or `^Xo` to choose whether the cursor keys should focus on browsing mode or the normal GDB command history.

C.4 Mastering the GDB Debugger

Advanced GDB use is a fine art that solves problems where other methods fail. After you have played with the basics of GDB, try returning here to understand why GDB is more powerful than most other debuggers.

- **Conditional Breakpoints:** `break [loc] if [cond]` — Stop only when a specific state is met.
- **Watchpoints:** `watch [var]` — Stop the instant a memory location changes.
- **GDBinit Scripts:** Automate your environment setup.

C.5 (Optional) Lab Exercise: Hunting the "Phantom" Write

C.5.1 The Scenario

You are working on a security-sensitive module. A local variable `isAdmin` is being overwritten by "phantom" data. There are no lines of code that explicitly assign a new value to `isAdmin`, yet its value changes during the execution of a loop.

C.5.2 The Buggy Code (`corrupt.c`)

```c
#include <stdio.h>

struct Session {
    int port_buffer[5];
    int isAdmin; // This should stay 0 (False)
};

int main() {
    struct Session user_session;
    user_session.isAdmin = 0;
    printf("Before loop: isAdmin = %d\n", user_session.isAdmin);
    // Simulating data processing
    for (int i = 0; i <= 5; i++) {
        user_session.port_buffer[i] = i + 100;
    }
    printf("After loop: isAdmin = %d\n", user_session.isAdmin);
    if (user_session.isAdmin != 0) {
        printf("SECURITY ALERT: Administrative privileges corrupted!\n");
    }
    return 0;
}
```

C.5.3 The Lab Steps

Step 1: Compilation Compile with debug symbols (`-g`) and disable optimizations (`-O0`) so the compiler doesn't reorder your code.

```
gcc -g -O0 corrupt.c -o corrupt
./corrupt
```

Observe the output. Why did isAdmin change to 105?

Step 2: Start the Debugger Launch the program inside GDB:

```
gdb ./corrupt
```

Step 3: Set the Watchpoint We need to stop the program the instant memory is modified.

1. Type `start` to stop at the beginning of `main`.
2. Type `watch user_session.isAdmin`.

Step 4: Execution and Analysis Type `continue`. GDB will pause execution and show you the exact line responsible for the change.

C.6 GDB Command Cheat Sheet

Category	Command	Description
Start	`run [args]`	Start execution from the beginning.
Control	`next` / `step`	Skip over / Step into functions.
Inspect	`print [var]`	Show current value of a variable.
Track	`watch [var]`	Stop the world when value changes.
Logic	`break [line] if [cond]`	Conditional breakpoint.
UI	`tui enable`	Show source code window (split view).

C.7 `.gdbinit` Automation Template

Save this as `.gdbinit` in your project folder to automate your setup.

But `.` in `.gdbinit` means this is a "hidden" file. Better yet, I prefer to save this as `gdbinit`, and then do:
`gdb -x gdbinit ./a.out`
If your `./a.out` command takes arguments, then use:
`gdb -x gdbinit --args ./a.out ARG1 ...`

```
# USAGE:  gdb -x THIS_FILE --args EXECUTABLE ARGS ...
# Stop GDB from interrupting gdbinit with questions
set breakpoint pending on
set pagination off
set confirm off

# Stop if about to exit prematurely
break _exit

# Never forget all the GDB commands that you used to get to the bug.
# Afterward, you can go:  gdb -x gdbinit_history
set history save on
set history filename gdbinit_history

# ============================================================================
# === If you're copying this gdbinit file, skip the optional parts, below. ===

# OPTIONALLY, add a new GDB command (useful only for this exercise).
define lab_setup
    start
    watch user_session.isAdmin
    printf "Environment ready. Watchpoint set.\n"
end
```

```
# OPTIONAL PRINT STYLES
# Print arrays with one element per line (in case of array of struct)
set print pretty on
# Print actual type of an object (if using C++)
set print object on
# If you would like GDB to to enable colors in output
set style enabled on

# OPTIONALLY, set it to execute immediately:
break main
run
```

C.8 (Optional) The Mystery Challenge: "The Broken Link"

A linked list of 1000 nodes is corrupted at node 742.

The Discovery Workflow:

1. **Run** until it crashes to find the "victim" ID (742).
2. **Restart** and set a **conditional breakpoint**: `break mystery.c:30 if curr->id == 742`.
3. **Continue** to jump straight to the relevant iteration.
4. **Watch** the pointer: `watch curr->next`.
5. **Continue** to see the exact function that changes the address to `0xBAD105`.

C.9 (Optional) The Art of Debugging Quiz

Instructions: Select the best answer for each question.

1. A program crashes at line 100 due to a NULL pointer assigned at line 20. Which is the manifestation?

- **A)** The NULL pointer assignment at line 20.
- **B)** The Segmentation Fault at line 100.
- **C)** The programmer's failure to initialize.
- **D)** The lack of an assert statement.
- **Hint:** *Think about which event is the observable symptom of the failure.*

2. What is the primary drawback of using print statements in large systems?

- **A)** They cannot display pointer values.
- **B)** They produce too much output, hiding relevant info.
- **C)** They automatically resolve the bug.
- **D)** They slow the program down to 0% speed.
- **Hint:** *Consider the "wall of text" problem mentioned in the lecture.*

3. An 'assert' statement is primarily used to:

- **A)** Automatically fix logic errors.
- **B)** Force the program to stop as close to the root cause as possible.
- **C)** Speed up execution.

- **D)** Convert C code to assembly.
- **Hint:** *Think about the goal of "stopping early" in the debugging process.*

4. Why use `perror()` after a failed system call?

- **A)** To hide the error from the user.
- **B)** To provide a human-readable description of why the OS call failed.
- **C)** To restart the system call.
- **D)** To bypass the debugger.
- **Hint:** *Recall the example:* `int rc = signal(...); if (rc == -1) { perror("signal"); }`.

5. Which GDB feature is best for finding 'who' corrupted a specific variable?

- **A)** Backtrace
- **B)** Conditional Breakpoint
- **C)** Watchpoint
- **D)** Next command
- **Hint:** *Think about the tool used in the "Phantom Write" lab.*

6. To skip 7,499 iterations of a loop and stop only at 7,500, you should use:

- **A)** `step 7500`
- **B)** `watch i == 7500`
- **C)** `break main.c:50 if i == 7500`
- **D)** `run 7500`
- **Hint:** *This saves you from hitting "continue" thousands of times.*

7. The purpose of a `.gdbinit` script is to:

- **A)** Compile the code.
- **B)** Automate setup, breakpoints, and custom commands.
- **C)** Prevent the program from crashing.
- **D)** Format the hard drive.
- **Hint:** *Consider the "fine art" of scripting your tools for efficiency.*

8. In GDB, the `step` command differs from `next` because:

- **A)** `step` enters function calls; `next` does not.
- **B)** `step` is for hardware; `next` is for software.
- **C)** `next` enters function calls; `step` does not.
- **D)** They are the exact same command.
- **Hint:** *Think about what happens when you reach a line like* `printf(...)`. *Do you want to go inside the library source code?*

9. Why compile with the `-g` flag?

- **A)** To make the program faster.
- **B)** To include debug symbols mapping machine code to source.
- **C)** To disable all print statements.
- **D)** To compress the executable.
- **Hint:** *Without this, GDB will only show you assembly code and memory addresses.*

10. What is a 'Heisenbug'?

- **A)** A bug that only occurs in German software.
- **B)** A bug that changes behavior when you try to observe it.
- **C)** A bug that is permanently fixed.
- **D)** A bug found only in assembly code.

- **Hint:** *This was mentioned in the section about why print statements can be problematic.*

C.10 Quiz Answer Key & Rationales

Q#	Answer	Rationale
1	**B**	The crash is the symptom (manifestation).
2	**B**	High volume of text creates "noise."
3	**B**	Early detection prevents error cascading.
4	**B**	Translates OS error codes into English.
5	**C**	Watchpoints monitor memory writes specifically.
6	**C**	Conditional breaks skip irrelevant states.
7	**B**	Scripting saves time and effort during repetitive sessions.
8	**A**	`step` goes deeper into the stack.
9	**B**	Without symbols, GDB cannot link binary to source lines.
10	**B**	Observation (like `printf`) alters timing and memory.