GDB ENGINEERING REFERENCE

Author: ddonle.com

Core Model

GDB controls a target process through **ptrace**. It can:

- Start, stop, step, and continue execution
- Inspect and modify memory, registers, and stack
- Set breakpoints, watchpoints, and catchpoints
- Attach to already-running processes
- Debug core dumps post-mortem

Build for debugging:
Always compile with `-g` to embed DWARF debug info.
Use `-O0` to disable optimizations (easier to follow).
Use `-O2 -g` for optimized builds (variables may be optimized out).

Typical session:
compile → launch gdb → set breakpoints → run → inspect → fix

Launching GDB

`gdb ./program` – load binary.
`gdb ./program core` – load binary with core dump.
`gdb -p PID` – attach to running process.
`gdb --args ./program arg1 arg2` – pass arguments at launch.
`gdb -batch -ex "run" -ex "bt" ./prog` – non-interactive scripted session.
`gdb -x script.gdb ./prog` – run commands from a file on start.
`gdb -tui ./prog` – launch with terminal UI (split source/command view).
Inside GDB, set program arguments:
`(gdb) set args arg1 arg2`
Quit GDB:
`(gdb) quit` or `q` (also `Ctrl-d`)

Running & Restarting

`run` / `r` – start the program (from the beginning).
`run arg1 arg2` – start with arguments.
`run < input.txt` – redirect stdin.
`start` – like `run` but stops at `main` automatically.
`starti` – stop at the very first instruction (before `main`).
`continue` / `c` – resume execution until next breakpoint.
`kill` – kill the running inferior.
`restart` – restart the program (same as `run` after first start).
`set follow-fork-mode child` – follow child on fork.
`set follow-fork-mode parent` – stay with parent on fork (default).
`set detach-on-fork off` – debug both parent and child simultaneously.

Breakpoints

`break main` / `b main` – break at function entry.
`break file.c:42` – break at line 42 of file.c.
`break *0x4005f0` – break at raw address.
`break func if x > 10` – conditional breakpoint.
`tbreak main` – temporary breakpoint (fires once, then deleted).
`rbreak regex` – set breakpoints on all functions matching regex.
`info breakpoints` / `i b` – list all breakpoints.
`delete 2` / `d 2` – delete breakpoint #2.
`delete` – delete all breakpoints.
`disable 2` / `enable 2` – disable or re-enable breakpoint.
`ignore 2 5` – skip breakpoint #2 the next 5 times.
`commands 2` – attach a list of commands to run when #2 fires:
`> print x   > continue   > end`
`save breakpoints file.gdb` – save breakpoints to file.
`source file.gdb` – reload saved breakpoints.

Watchpoints & Catchpoints

Watchpoints – break when a value changes:
`watch x` – break when variable `x` is written.
`rwatch x` – break when `x` is read.
`awatch x` – break when `x` is read or written.
`watch *(int*)0xdeadbeef` – watch a raw memory address.
`watch x if x > 100` – conditional watchpoint.

Catchpoints – break on events:
`catch syscall` – break on any syscall.
`catch syscall write` – break on `write` syscall.
`catch throw` – break when a C++ exception is thrown.
`catch catch` – break when a C++ exception is caught.
`catch fork` – break on `fork()`.
`catch exec` – break on `exec()`.

Stepping

`next` / `n` – step over: execute one line, don't enter calls.
`step` / `s` – step into: enter function calls.
`nexti` / `ni` – step over one machine instruction.
`stepi` / `si` – step into one machine instruction.
`finish` / `fin` – run until current function returns, print return value.
`until` / `u` – run until current line is passed (exits loops).
`until file.c:55` – run until a specific location.
`advance func` – run until reaching `func` (like a one-shot breakpoint).
`jump *0x4005f0` – jump execution to address (no stack change).
`signal SIGINT` – deliver a signal to the inferior.

Repeat last step command by pressing **Enter**.

Stack Inspection

`backtrace` / `bt` – print full call stack.
`bt full` – print stack with local variables at each frame.
`bt 5` – print only the innermost 5 frames.
`frame 3` / `f 3` – switch to stack frame #3.
`up` / `down` – move one frame up or down the stack.
`info frame` – show details of the current frame.
`info args` – show current function's arguments.
`info locals` – show local variables in the current frame.
`where` – alias for `backtrace`.

Inspecting Variables & Expressions

`print x` / `p x` – print value of expression `x`.
`print/x x` – print in hex. `print/t x` – binary. `print/c x` – char.
`print/d x` – decimal. `print/f x` – float. `print/a x` – address.
`print *ptr` – dereference pointer.
`print arr[0]@10` – print 10 elements of array from `arr[0]`.
`print $rax` – print register value.
`display x` – auto-print `x` after every step.
`undisplay 1` – remove auto-display #1.
`info display` – list active auto-displays.
`whatis x` – show type of expression.
`ptype MyStruct` – show full type definition.
`set variable x = 42` – modify a variable's value at runtime.

Memory Inspection

`x/NFU addr` – examine memory at `addr`.

N	repeat count
F	format: `x` hex, `d` decimal, `s` string, `i` instruction
U	unit: `b` byte, `h` halfword, `w` word, `g` giant (8B)

`x/10xw 0xdeadbeef` – 10 hex words at address.
`x/s 0x4006a0` – print null-terminated string at address.
`x/20i $pc` – disassemble 20 instructions from current PC.
`x/4gx $rsp` – show top 4 stack slots (64-bit).
`info address sym` – show address of a symbol.
`info symbol 0x4005f0` – show symbol at address.

Registers

`info registers` / `i r` – show all general-purpose registers.
`info all-registers` – show all registers including FPU/SSE.
`info registers rax rbx rsp` – show specific registers.
`print $rip` – print instruction pointer.
`print $rsp` – print stack pointer.
`set $rax = 0` – modify a register value.

Common x86-64 registers:

`$rip`	instruction pointer
`$rsp`	stack pointer
`$rbp`	base (frame) pointer
`$rax`	return value / accumulator
`$rdi $rsi $rdx`	1st, 2nd, 3rd arguments

Disassembly

`disassemble` / `disas` – disassemble current function.
`disas func` – disassemble named function.
`disas 0x400500,0x400550` – disassemble address range.
`disas/m func` – interleave source lines with assembly.
`disas/r func` – show raw bytes alongside instructions.
`set disassembly-flavor intel` – use Intel syntax (default is AT&T).
`set disassembly-flavor att` – switch back to AT&T syntax.
`layout asm` – TUI split view showing assembly.
`layout src` – TUI split view showing source.
`layout regs` – TUI view showing registers.

Source Navigation

`list` / `l` – show 10 lines around current location.
`list func` – show source around function.
`list file.c:42` – show source around line 42 in file.
`list -` – show previous 10 lines.
`set listsize 20` – change lines shown by `list`.
`directory /path/to/src` – add source search directory.
`info source` – show current source file info.
`info sources` – list all known source files.
`search pattern` – search forward in source for pattern.
`tui enable` – enable terminal UI mode.
`Ctrl-x Ctrl-a` – toggle TUI mode on/off.
`Ctrl-x 2` – cycle TUI layout (src / asm / regs).

Threads

`info threads` – list all threads with IDs and current location.
`thread 3` / `t 3` – switch to thread #3.
`thread apply all bt` – print backtrace for every thread.
`thread apply all bt full` – full backtrace with locals for all threads.
`thread apply 1 2 print x` – apply command to specific threads.
`set scheduler-locking on` – only current thread runs when stepping.
`set scheduler-locking off` – all threads run (default).
`set scheduler-locking step` – lock during step, release on continue.
`break func thread 2` – breakpoint only for thread #2.

Core Dumps & Post-Mortem

Generate a core dump:
`ulimit -c unlimited` – enable core dumps in shell.
`echo core > /proc/sys/kernel/core_pattern` – set core file path.
Crash the program or `kill -ABRT PID` to force a core dump.

Analyze a core dump:
`gdb ./program core` – load program and core file.
`bt` – see where the crash occurred.
`info registers` – inspect register state at crash.
`x/20i $rip-40` – disassemble around the crash point.

Generate core from inside GDB:
`generate-core-file` – write core of current inferior to disk.

Reverse Debugging

Record execution and step **backwards** through it.
`record` – start recording execution (process replay log).
`record stop` – stop recording.
`reverse-continue` / `rc` – run backwards to previous breakpoint.
`reverse-next` / `rn` – step backwards over a line.
`reverse-step` / `rs` – step backwards into a function.
`reverse-finish` – go back to where the current function was called.
`set exec-direction reverse` – make `continue`/`next`/`step` go backwards.
`set exec-direction forward` – restore normal direction.

Recording is slow; use on small reproducible sections. Requires `record full` or `record btrace` (hardware-assisted).

Scripting & Automation

GDB command files (`.gdb`):

```
# script.gdb
set pagination off
break main
run
bt
quit
```

`gdb -batch -x script.gdb ./prog`

Python scripting inside GDB:
`python print(gdb.parse_and_eval("x"))` – evaluate expression.
`python gdb.execute("bt")` – run GDB command from Python.
`source myscript.py` – load a Python GDB extension.

~/.gdbinit – auto-loaded on startup:

```
set pagination off
set print pretty on
set disassembly-flavor intel
set history save on
```

Pretty Printing & Display

`set print pretty on` – indent nested structures.
`set print array on` – print arrays one element per line.
`set print array-indexes on` – show array indices.
`set print null-stop on` – stop char array print at null byte.
`set print object on` – show derived type of C++ objects.
`set print vtbl on` – show C++ virtual table info.
`set print demangle on` – demangle C++ symbol names (default on).
`set print elements 100` – limit elements printed (0 = unlimited).
`set print repeats 0` – always print repeated elements.
`set pagination off` – disable `---Type <return>---` prompts.

Pretty-printers for STL / Python:
Install GDB pretty-printers for libstdc++, Boost, etc. via `source` in ~/.gdbinit to get readable `std::vector`, `std::map` output.

Remote & Embedded Debugging

gdbserver – run on the target machine:
`gdbserver :1234 ./program` – listen on port 1234.
`gdbserver :1234 --attach PID` – attach to running process.
GDB client – run on the host:
`target remote host:1234` – connect to gdbserver.
`target extended-remote host:1234` – persistent connection (allows `run`).
`load` – flash/load program onto target.
`monitor reset` – send reset command to target (firmware-specific).

OpenOCD / JTAG:
`target extended-remote localhost:3333` – connect via OpenOCD.
`set architecture arm` – set target architecture manually.

Checkpointing

Checkpoints save the exact state of the inferior and allow returning to it.
`checkpoint` – save current execution state.
`info checkpoints` – list saved checkpoints.
`restart N` – restore inferior to checkpoint #N.
`delete checkpoint N` – remove checkpoint #N.

Useful for exploring multiple code paths from one point without restarting. Unlike `record`, checkpoints fork the process rather than replaying — faster but uses more memory.

Large-Scale Debugging Practices

- Always compile with `-g`; keep a debug build separate from release.
- Use `set pagination off` and `set print pretty on` in ~/.gdbinit.
- Save and `source` breakpoint files for complex sessions.
- Use `thread apply all bt` as first step in any threading bug.
- Use `watch` instead of `printf` debugging for data corruption.
- Prefer `catch throw` over guessing where exceptions originate.
- Use `-batch` mode in CI to run automated crash triage.
- Use `gdbserver` + cross-GDB for embedded and container targets.
- Install libstdc++ pretty-printers for readable STL containers.
- Use `record` + reverse stepping to find the *cause*, not just the crash.

🟦 Core concepts / inspection
🟧 Breakpoints / watchpoints
🟥 Core dumps / danger
■ Threads / remote
🟪 Scripting / display config
🟩 Stepping / workflow