GDB Step-by-Step Guide (practical recipes + commands)

0) Prep: compile with debug info

Always compile with debug symbols and reasonable optimization level for debugging:

```
gcc -g -O0 -fno-omit-frame-pointer -Wall -Wextra -o myprog myprog.c
# or for C++
g++ -g -O0 -fno-omit-frame-pointer -Wall -Wextra -o myprog myprog.cpp
```

Notes:

- `-g` = include debug symbols.
- `-O0` avoids many compiler optimizations that make stepping & variables confusing.
- For diagnosing performance/optimization issues you might use `-O2` but be prepared for optimized frames and missing local variables.

---

1) Start GDB & simple run

```
gdb ./myprog
(gdb) set pagination off            # optional: so outputs don't stop with --
More--
(gdb) run                            # or `r arg1 arg2`
```

Or start with args from shell:

```
gdb --args ./myprog arg1 arg2
(gdb) run
```

Useful startup options:

- `gdb -q ./myprog` (quiet)
- `start` — run until `main()` and stop there.

---

2) Breakpoints (basic → advanced)

Set a breakpoint and control it:

```
(gdb) break main                    # break at function main
(gdb) break file.c:123              # break at line 123 in file.c
(gdb) break myfunc                  # break at start of myfunc
(gdb) break *0x4005d0               # break at address
(gdb) tbreak somefunc               # temporary breakpoint (auto delete once
hit)
(gdb) info breakpoints               # show all breakpoints with numbers
(gdb) disable 2                      # disable bp #2
(gdb) enable 2                       # enable bp #2
```

```
(gdb) delete 2                      # delete bp #2
(gdb) condition 3 i>100             # bp#3 only when i>100
```

Quick: `b` = `break`. Example:

```
(gdb) b myfile.c:45 if x==0
```

---

3) Stepping & running control

```
(gdb) next          # n  — next source line (step over)
(gdb) step          # s  — step into functions
(gdb) finish        # run until current function returns
(gdb) continue      # c  — continue to next breakpoint
(gdb) until 200     # run until line 200 in current file
(gdb) jump 120      # jump execution to line 120 (dangerous)
(gdb) interrupt     # Ctrl-C to pause a running program
```

---

4) Inspecting variables, types & expressions

```
(gdb) print x               # p x
(gdb) print/x x             # print hex
(gdb) print *(mystruct *)ptr
(gdb) ptype var              # show type of var
(gdb) info locals            # show local variables in current frame
(gdb) info args              # show function arguments
(gdb) set var i=42           # change variable i to 42
(gdb) display i              # auto-print i each stop
(gdb) undisplay 1            # remove displayed entry #1
```

Formatting examples:

```
(gdb) print /x myint         # print in hex
(gdb) print /s mycharptr     # print string
(gdb) x/16xb &buffer         # examine memory: 16 bytes, hex, byte-size
```

`x/NFMT ADDRESS` — generic memory examine:

- `N` = count, `F` = format (`x` hex, `d` decimal, `s` string, `i` instruction), `T` = size (`b` byte `h` halfword `w` word `g` 8 bytes).

Example:

```
(gdb) x/32xb buf   # show 32 bytes starting at buf in hex
(gdb) x/8g 0x7fffdeadbeef  # show 8 8-byte words
```

---

5) Backtrace, frames, and stack inspection

```
(gdb) backtrace          # bt    -- show call stack
(gdb) bt full            # show locals for each frame
(gdb) frame 3            # go to frame 3
(gdb) info frame         # info about current frame
(gdb) info registers     # show CPU registers (useful for crashes)
```

If you compiled without frame pointers, `bt` may be less reliable for optimized builds.

---

6) Crash analysis (segfaults, aborts)

When program crashes (during `run`) GDB will stop and show signal:

```
Program received signal SIGSEGV, Segmentation fault.
0x00005555555541b2 in foo () at file.c:27
27      *p = 0;
(gdb) bt
(gdb) frame 0
(gdb) info locals
(gdb) print p
(gdb) p/x $rip    # on x86_64, $rip is instruction pointer
(gdb) disassemble $pc-32, $pc+32
```

If you have a core dump:

```
gdb ./myprog core
(gdb) bt full
(gdb) info registers
(gdb) info threads        # if core contains multiple threads
```

Checklist for crash:

1. `bt full` to see stack & locals.
2. Inspect pointer variables in the crashing frame (e.g., `p ptr`, `p *ptr`).
3. `disassemble` around `$pc`/`$rip`.
4. If assembly confusing, use `list` to show source around offending line: `list 20,40`.

---

7) Core dump debugging

Generate a core by running under shell with `ulimit -c unlimited`:

```
ulimit -c unlimited
./myprog
# if it crashes, find core file (core or core.<pid>)
gdb ./myprog core
(gdb) bt
```

Helpful commands:

```
(gdb) info proc mappings   # show memory map (linux only, requires gdb attached
to running process or core)
(gdb) info threads
(gdb) thread apply all bt  # backtrace for all threads
```

---

8) Multithreaded debugging & locks / deadlocks

Primary tools for threads:

```
(gdb) info threads                 # list threads with id & status
(gdb) thread <n>                   # switch to thread n (use gdb thread id)
(gdb) thread apply all bt         # get backtraces of all threads
(gdb) thread apply all bt full    # include locals
```

Finding deadlocks / lock contention:

1. Stop the program with Ctrl-C or wait until it's hanging.
2. `thread apply all bt` to see where each thread is blocked — look for calls like `pthread_mutex_lock`, `futex`, `sem_wait`, `pthread_cond_wait`, or long sleeps.
3. Switch into frames that call lock functions and inspect mutex variables:

   ```
   (gdb) frame 3
   (gdb) print lock_var
   (gdb) p/x *(pthread_mutex_t *)&lock_var
   ```

4. If using higher-level locks (std::mutex), examine corresponding internal fields or backtrace to find who holds the lock.

Tip: run `thread apply all bt` and look for the stack frames of the lock owner. Often the thread holding the mutex will be visible in another thread's backtrace.

Extra: GDB's `info locks` is not a universal command — availability depends on GDB version and platform. If present, it lists known locks.

---

9) Watchpoints & data breakpoints

Use watchpoints to stop when a memory location changes:

```
(gdb) watch myvar              # stop when myvar changes
(gdb) rwatch myvar             # stop on read of myvar
(gdb) awatch myvar             # stop on read or write
(gdb) info watchpoints
```

Notes:

- Hardware watchpoints are limited (usually 4 on x86). If too many, GDB will fallback to software watchpoints, which are slower.
- Use conditional watchpoints: `watch myvar if i>100`.

---

10) Remote debugging (gdbserver) & embedded

On target (remote) machine:

```
gdbserver :1234 ./myprog arg1 arg2
# or to attach: gdbserver :1234 --attach <pid>
```

On host:

```
gdb ./myprog            # the host binary (with symbols)
(gdb) target remote target-ip:1234
(gdb) continue
```

Useful:

- `set solib-search-path /path/to/target/libs` — set where to load shared libraries.
- `set sysroot /path/to/sysroot` — useful for cross-debugging.

Embedded: often use `arm-none-eabi-gdb` + OpenOCD:

```
(gdb) target remote localhost:3333   # OpenOCD port
(gdb) monitor reset halt
(gdb) load                            # download program to target
```

---

11) Recording & reverse debugging

GDB supports process recording (useful to "reverse" step). Commands:

```
(gdb) target record-full            # start recording the process
(gdb) record instruction-history    # older syntax
# then reverse execution:
(gdb) reverse-continue
(gdb) reverse-step
(gdb) reverse-next
(gdb) record stop
```

Notes:

- Recording can be slower and use lots of disk. Not all platforms support all features.
- If your version lacks reverse commands, consider rr (https://rr-project.org/) — a specialized tool for reproducible recording/replay (val/pointer).

---

12) Memory leak detection (what to use)

Short answer: GDB is not ideal for leak detection. Use specialized tools:

- **Valgrind** (`valgrind --leak-check=full ./myprog`) — classical tool to find leaks.
- **AddressSanitizer (ASan)**: compile with `-fsanitize=address -g -O1` and run — it prints leak/detects use-after-free.
- **LeakSanitizer (LSan)** or `-fsanitize=leak`.
- For large programs, `massif` (valgrind) for heap profiling.

You *can* combine GDB and ASan:

- Run program compiled with ASan under GDB. ASan prints diagnostics to stderr and then you can `bt` in GDB when it aborts:

```
g++ -g -O1 -fsanitize=address -fno-omit-frame-pointer myprog.cpp -o myprog
gdb ./myprog
(gdb) run
# when ASan aborts, do (gdb) bt
```

If you must hunt leaks with GDB only:

1. Reproduce high memory usage.
2. Use `info proc mappings` (Linux) to see heap segments.
3. Examine `malloc` internals if you know the allocator/internals (not beginner-friendly).
   → For practical leak hunting, prefer Valgrind/ASan.

---

13) Debugging optimized code

Optimized builds behave oddly (missing locals, inlined functions). Tips:

- Use `-Og` if you want some optimization but still debuggable.
- Use `volatile` for variables you must preserve for debugging (temporary trick).
- Use `set print object on` and `set print pretty on` for C++ objects.
- Use `disassemble` + `info line` to map instructions to source lines:

```
(gdb) info line 42         # which address corresponds to line 42
(gdb) disassemble 0x400abc,0x400bcd
```

---

14) GDB scripts & automation

Create `.gdbinit` in your project or home directory for default settings:

```
set pagination off
set print pretty on
define hook-run
  echo Starting program -- breakpoints:\n
  info breakpoints
end
```

You can also feed commands:

```
gdb -x my_gdb_commands.txt ./myprog
```

---

15) Useful debugging one-liners / cheats

- Run until a function returns:

  ```
  (gdb) break myfunc
  ```

```
(gdb) commands  # run when breakpoint hit
> finish
> continue
> end
```

- Print backtrace of all threads:

```
(gdb) thread apply all bt
```

- Attach to running process:

```
sudo gdb -p <pid>   # attach by PID (requires permissions)
```

- Run program and stop on malloc failure (example using `catch`):

```
(gdb) catch throw      # for C++ exceptions
(gdb) catch syscall    # where supported
```

---

16) Practical workflows (recipes)

A — General interactive debugging (logic bug)

1. Compile with `-g -O0`.
2. Start GDB: `gdb --args ./myprog a b`
3. `b main`, `run`
4. Step to suspicious function: `b suspicious_func`, `c`
5. At breakpoint: `bt`, `info locals`, `p var`, `n` / `s`.
6. Add conditional break: `condition 4 i>100`
7. When fixed, `set var` to test alternate values without recompiling.

B — Crash (segfault)

1. `gdb ./myprog core` (if core exists) or run under gdb: `gdb ./myprog` → `run`.
2. When it stops: `bt full`, `frame 0`, `info locals`, `p ptr`.
3. If pointer NULL, backtrace to see where it was set. Use `reverse-step` if recorded.

C — Memory leak (use ASan/Valgrind)

1. Compile with ASan: `-fsanitize=address -g -O1`.
2. Run: `./myprog` — check ASan output for leak stack traces.
3. Or `valgrind --leak-check=full ./myprog`.
4. Use `gdb` to inspect suspect code once you have the leak stack trace.

D — Deadlock / stuck threads

1. If process hung, attach with `gdb -p <pid>` or run under gdb and Ctrl-C.
2. `thread apply all bt` — see where each thread is blocked.
3. Identify thread that owns mutex (look at frames) and inspect its stack to find where it took the lock.
4. Fix by ensuring consistent lock ordering or adding timeouts.

E — Remote target / embedded

1. Start `gdbserver` on target: `gdbserver :1234 ./myprog`
2. On host: `gdb ./myprog` → `target remote target:1234`
3. Use `load` to upload binary (if supported), `monitor` commands for OpenOCD.

---

17) Debugging C++ specifics

- Pretty printers: enable libstdc++ printers for STL containers (usually auto-enabled).
- `p vec` prints vector metadata; `p vec._M_impl._M_start[0]` for elements (implementation-dependent).
- Use `catch throw` to break when exceptions are thrown:

```
(gdb) catch throw
(gdb) catch catch   # break at catch?
```

- Use `info threads` + `thread apply all bt` for exceptions crossing threads.

---

18) Tips & gotchas

- If symbols don’t appear: check binary actually contains debug info (`file myprog` or `readelf -S myprog | grep debug`).
- Shared libraries: use `set breakpoint pending on` so breakpoints in not-yet-loaded SOs are accepted, and `info sharedlibrary`.
- Use `set follow-fork-mode child` if the program forks and you want to debug the child.
- `set detach-on-fork off` to keep both processes attached (advanced).
- For aggressive optimization, add frame pointers: `-fno-omit-frame-pointer` to improve backtraces.
- Always recompile after changing source; GDB can edit slightly with `set var`, but code changes need rebuild.

---

19) When GDB isn’t enough

- **Memory leaks / heap corruption:** Valgrind, ASan, LSAN, jemalloc's profiling tools.
- **Heisenbugs / timing-sensitive races:** rr, rr + gdb, or Thread Sanitizer (`-fsanitize=thread`).
- **Large production issues:** capture core, use logging, or gdbserver attach.

---

20) Short quick reference (commands)

- start/run: `run`, `start`
- break: `break`, `tbreak`, `condition`
- step: `step`, `next`, `finish`
- continue: `continue`
- inspect: `print`, `ptype`, `info locals`, `info args`
- memory: `x/...`, `display`
- stack: `bt`, `frame`, `info frame`
- threads: `info threads`, `thread <n>`, `thread apply all bt`
- watch: `watch`, `rwatch`, `awatch`
- remote: `target remote host:port`
- core: `gdb ./prog core`
- exit: `quit`