Lecture 15: Debugging Without a Debugger & GDB Deep Dive

1. Debugging Without a Debugger

1.1 Static Checking

- **Static checking** is performed at compile-time using tools like `gcc` and `clang`.
- **Other static tools:**
  - **Linters:** e.g., `cpplint`, `clang-tidy` for style and bug patterns.
  - **Static analyzers:** e.g., Coverity, Clang Static Analyzer, Infer (Facebook), which can find deep bugs across function boundaries.
  - **Formal verification:** Mathematical proofs of correctness (rare in practice, used in safety-critical systems).
- **Limitations:**
  - Cannot catch all bugs (e.g., runtime-specific issues).
  - Theoretical limits (e.g., Halting Problem) prevent perfect static analysis.
  - May produce false positives/negatives.

Tool/Method	What It Checks	Example Tool
Compiler Warnings	Syntax, types, some logic	`-Wall`, `-Wextra`
Linter	Style, simple bug patterns	`cpplint`, `pylint`
Static Analyzer	Deep logic, interprocedural	Clang Static Analyzer, Coverity
Formal Verification	Mathematical correctness	Frama-C, SPARK

1.2 Dynamic Checking

- **Dynamic checking** occurs during program execution.
- Catches bugs missed by static checking, but only for the specific run/test case.
- **Manual dynamic checks:**
  - Add assertions to validate state at runtime.
  - Example: Array bounds checking

    ```
    if (!(0 <= i && i < n)) error();
    a[i];
    ```

  - Example: Integer overflow (incorrect check)

    ```
    if (j * k > INT_MAX) error(); // This does NOT work in C/C++
    ```

    - In C/C++, signed integer overflow is undefined behavior. The implementation can do anything, so the check above is unreliable.
  - **Correct overflow check (C23/C++26):**

    ```
    #include <stdckdint.h>
    if (__builtin_mul_overflow(j, k, &i)) error();
    ```

    - `__builtin_mul_overflow` returns true if overflow occurred.
    - C++26 will have similar utilities.
  - **Downsides:**
    - Tedious and error-prone to add checks everywhere.
    - Easy to make mistakes in the checks themselves.

1.2.1 GCC Sanitizer Flags

- Compiler flags that insert runtime checks for various classes of bugs:

Flag	Purpose	Captures
`-fsanitize=undefined`	Catch undefined behavior (e.g., overflow)	Integer overflow, divide by 0
`-fsanitize=address`	Catch memory/address issues	Buffer overflows, bad pointers
`-fsanitize=thread`	Detect race conditions	Multi-thread concurrency bugs

Flag	Purpose	Captures
`-fsanitize=leak`	Find memory leaks	Leaked malloc() allocations

- **Limitations:**
  - Some flags cannot be used together (e.g., `undefined` vs `address`).
  - Not all undefined behaviors are caught (e.g., some pointer aliasing bugs).
  - These flags slow down execution.
  - Only effective if the code path is executed during testing.

1.2.2 Valgrind

- Tool for dynamic analysis at the binary level (no recompilation needed).
- Detects memory errors, leaks, and some undefined behaviors.
- **Advantages:**
  - Can be used on production binaries.
  - No need for special compilation flags.
- **Disadvantages:**
  - Much slower than sanitizer flags (interprets each instruction).
  - Lacks source code context, so may miss semantic errors.
  - Cannot catch all concurrency bugs or logic errors.

Tool	Source Required	Speed Impact	Coverage	Usability in Production
Sanitizers	Yes	Moderate	Targeted checks	No
Valgrind	No	High	Broader, shallower	Yes
Fuzzers	Yes	High	Randomized input paths	No

1.2.3 Compiler Flag: `-fwrapv`

- Forces signed integer overflow to wrap around (modulo 2^n).
- Makes code more predictable, but disables some optimizations (e.g., loop unrolling).
- **Not the default** because it can slow down code and prevent optimizations.

Example: Loop Unrolling

```
for (int i = 0; i < n; i++) {
  f(i);
}
```

- If `n` is `INT_MAX`, `i++` might overflow.
- With `-fwrapv`, the compiler cannot assume overflow doesn't happen, so it can't unroll the loop.

Table: Static vs. Dynamic Checking

Aspect	Static Checking	Dynamic Checking
When Performed	Compile-time	Run-time
Tools	Compiler, linters, analyzers	Assertions, sanitizers, Valgrind
Bugs Detected	Syntax, type, some logic	Memory, concurrency, runtime
Overhead	None at runtime	Slows down execution
Limitations	Can't catch runtime-specific	Only checks tested code paths

More Dynamic Tools

- **Fuzzers:** e.g., AFL, libFuzzer. Generate random/semi-random inputs to find crashes and bugs.
- **Dynamic analyzers:** e.g., Dr. Memory, ThreadSanitizer (for race conditions).

Assertions: Static vs. Runtime

- **Runtime assertions:** Checked during execution. Can be compiled out with `-DNDEBUG`.
- **Static assertions:** Checked at compile time (e.g., `static_assert` in C11/C++11).
- **Pitfall:** Assertions with side effects can change program behavior if compiled out.

```
// BAD: Side effect in assertion
assert(x++ > 0); // x is incremented only if assertions are enabled!
```

Assertion Type	When Checked	Can Be Compiled Out?	Example
Runtime	Run-time	Yes (`-DNDEBUG`)	`assert(x > 0);`
Static	Compile-time	No	`static_assert(...);`

Practical Debugging Workflow

Suppose a program crashes with a segmentation fault. A robust workflow:

```
[Start]
  |
  v
[Reproduce Failure]
  |
  v
[Static Checking (compiler, linter)]
  |
  v
[Dynamic Checking (sanitizer, Valgrind, fuzzer)]
  |
  v
[Add Assertions/Logging]
  |
  v
[Trace Error (print/debugger)]
  |
  v
[Fix Bug]
  |
  v
[Retest All Cases]
  |
  v
[End]
```

- **Decision Tree Example:**
  - If bug is a crash: Use sanitizer/Valgrind first.
  - If bug is a logic error: Add assertions and logging.
  - If bug is intermittent: Try ThreadSanitizer or fuzzing.

Debugging Tool Comparison Table

Tool/Method	Detects	Overhead	Source Needed	Best For
Compiler Warnings	Syntax, types	None	Yes	Early bug detection
Linter	Style, simple bugs	None	Yes	Code quality
Static Analyzer	Deep logic	Low	Yes	Complex bugs
Sanitizer	Memory, UB, races	Med	Yes	Memory/concurrency bugs
Valgrind	Memory, leaks	High	No	Production binaries
Fuzzer	Crashes, edge cases	High	Yes	Unusual input bugs

2. Portability Checking

- Ensures code runs correctly on different platforms (OS, architecture, browser, etc.).
- **Examples:**
  - Cross-browser JavaScript testing.
  - Cross-platform compilation (32-bit vs 64-bit).
  - Testing with different OS/browser/plugin combinations.
- **Techniques:**
  - Build and run on multiple environments.
  - Use compiler flags like `-m32` to generate 32-bit binaries.
  - **Feature testing:** Use tools like Autoconf, CMake to check for features, not just OS.

Flag	Purpose
`-m32`	Force 32-bit compilation on GCC
default	Typically compiles to 64-bit binary

- **Note:** Portability checking can be expensive due to the combinatorial explosion of possible environments.
- **Best Practice:** Prefer feature checks over OS checks. Example:

  ```
  #ifdef HAVE_RENAMEAT2
    // Use renameat2
  #else
    // Fallback
  #endif
  ```

Approach	Pros	Cons
OS-based checks	Simple, direct	Brittle, not future-proof
Feature checks	Robust, portable	More setup, needs tooling

3. Test Cases

- **Purpose:** Not to prove code works, but to find bugs.
- **Mindset:** "If my test cases didn't find a bug, I failed."
- **Types of tests:**
  - **Unit tests:** Test individual functions/components.
  - **Integration tests:** Test interactions between components.
  - **Regression tests:** Ensure old bugs stay fixed.
  - **Fuzz tests:** Randomized input to find edge cases.
- **Test infrastructure:**
  - Automate test execution (shell scripts, `make check`, CI systems).
  - Run tests in parallel (`make -j N`).
  - Separate quick/cheap tests (run frequently) from heavy/expensive tests (run less often).
- **Tools:**
  - Scripts (e.g., `run_tests.sh`)
  - Makefiles with `check` targets
  - GitHub Actions/CI
- **Test case generation:**
  - LLMs (Large Language Models) are effective for generating test cases (Meta found 40–50% LLM contribution is optimal).
- **Randomness testing:**
  - Impossible to mathematically prove randomness.
  - Can test statistical properties (bit balance, lack of patterns, distribution coverage).
  - Example of a bad random generator that passes naive tests:

    ```
    return UINT64_MAX / 3; // Alternating bits
    ```

- **Test Coverage:**
  - **Statement coverage:** Every line executed.
  - **Branch coverage:** Every branch taken.
  - **Path coverage:** Every possible path (usually infeasible for large programs).

Test Type	Scope	Detects
Unit	Function	Local logic errors
Integration	Subsystem	Interface bugs
Regression	Whole system	Recurring bugs
Fuzz	Whole system	Edge cases, crashes

4. Defensive Programming

- **Goals:**
  - Prevent bugs before they occur.
  - Minimize the impact of bugs that do occur.

    - Detect bugs early and reliably.
- **Techniques:**
    - Static checking (compiler warnings)
    - Dynamic checking (sanitizers, Valgrind)
    - Test case design for failure
    - Defensive coding (asserts, bounds checks)
    - **Input validation:** Always check user input for validity.
    - **Fail-fast:** Abort early on error to avoid propagating bad state.
    - **Error handling patterns:** Return error codes, use exceptions, or error objects as appropriate.

Defensive Technique	Example/Pattern	Benefit
Input validation	`if (!valid(x)) return -1;`	Prevents bad data
Assert invariants	`assert(ptr != NULL);`	Catches bugs early
Fail-fast	`exit(1)` on error	Avoids cascading failures
Error codes	`return -1;`	Explicit error propagation
Exception handling	`try { ... } catch { ... }`	Structured error management

5. Terminology: Error, Fault, Failure

Term	Description	Real-World Analogy
Error	Developer's mistake (mental/conceptual)	Forgetting to check tire pressure
Fault	Error reflected in the code (latent bug)	Flat tire in the car
Failure	Fault triggered during execution (observable bug)	Car accident

- **Debugging process:**
    - Start with symptoms (failure), trace back to fault (code), then to error (developer's mistake).

```
[Error: Developer's Mistake]
           |
           v
   [Fault: Bug in Code]
           |
           v
[Failure: Runtime Symptom]
           |
           v
[Debugging: Trace Back]
           |
           v
[Fix: Correct the Error]
```

Step	What You See	What It Means
Failure	Crash, wrong output	Something went wrong
Fault	Bug in code	Root cause in implementation
Error	Mental slip	Design/logic misunderstanding

6. Debugging Best Practices

6.1 Steps

1. **Reproduce/Stabilize the failure**
    - Make the bug consistent and repeatable.
    - May require disabling features like ASLR (Address Space Layout Randomization).
2. **Locate the fault**
    - Use backwards reasoning from symptoms to code.
    - Use debugger features to narrow down the cause.
3. **Minimize the test case**
    - Reduce input to the smallest case that still triggers the bug.
4. **Form a hypothesis**
    - Based on evidence, guess the root cause.

5. **Test the hypothesis**
   - Change code or add diagnostics to confirm/refute.
6. **Fix and retest**
   - Correct the bug and rerun all tests to ensure the issue is resolved.

```
[Observe Failure]
      |
      v
[Reproduce & Stabilize]
      |
      v
[Minimize Test Case]
      |
      v
[Form Hypothesis]
      |
      v
[Test Hypothesis]
      |
      v
[Fix & Retest]
```

6.2 Anti-patterns

- Randomly modifying code lines hoping it works (futile, non-scalable).
- Avoid using GDB as a crutch; it's for reasoning, not fixing.
- Ignoring compiler warnings.
- Not automating tests (manual testing is error-prone).
- Failing to minimize test cases (harder to debug).

Debugging Checklist

- [ ] Can you reproduce the bug reliably?
- [ ] Have you checked compiler warnings and static analysis?
- [ ] Have you run dynamic tools (sanitizer, Valgrind, fuzzer)?
- [ ] Have you minimized the test case?
- [ ] Do you understand the code path leading to the bug?
- [ ] Have you confirmed the fix with all relevant tests?

7. GDB: Debugger Deep Dive

7.1 GDB's Role

- GDB is a "Program Execution Explorer."
- GDB controls the debugged process by communicating with the OS kernel.
- Can start a program, attach to a running process, or modify program state.

7.2 Key GDB Commands

Command	Purpose
`run` / `r`	Start the program within GDB
`quit` / `q`	Exit GDB
`attach PID`	Attach to an already running process
`detach`	Detach from the debugged process

Setup Commands

Command	Description
`set cwd /path`	Set working directory for debugged proc
`set env VAR value`	Set environment variable
`set disable-randomization off`	Enable ASLR

- **ASLR (Address Space Layout Randomization):**
  - Randomizes memory layout to prevent exploits.

    - Hurts reproducibility for debugging.
    - GDB disables ASLR by default for reproducibility.

7.3 Breakpoints

Command	Description
`break <loc>` / `b`	Set breakpoint at function or line
`info break` / `ib`	List all breakpoints
`delete <num>` / `d`	Remove specified breakpoint
`cond <num> <expr>`	Set condition for breakpoint (advanced)

- **Implementation:**
    - GDB replaces the instruction at the breakpoint with a trap instruction.
    - When the program hits the trap, it stops and GDB regains control.
    - **Hardware breakpoints:** Use CPU support, limited in number (e.g., 4 on x86-64).
    - **Software breakpoints:** Unlimited, but slower (require code modification).

7.4 Control Commands

Command	Action
`continue` / `c`	Resume execution after break
`step` / `s`	Step into the next line of source code (includes functions)
`next` / `n`	Step over function calls
`stepi`	Step a single machine instruction
`finish`	Run until current function returns

- **Note:** Stepping can be confusing with optimized code; use `-O0` and `-g3` for best results.

7.5 Advanced Commands

Command	Purpose
`reverse-continue` / `rc`	Execute backwards to previous state (requires special setup, slows down)
`watch <expr>`	Set a watchpoint to break when expression value changes
`checkpoint` / `restart`	Save and reload program state (manual reverse execution)
`print <expr>` / `p`	Evaluate and print a variable or expression
`define <macro>`	Define a custom GDB macro (automation)

- **Reverse execution:**
    - `rc` requires GDB to keep snapshots of program state, which is slow and memory-intensive.
    - `checkpoint`/`restart` is a manual, more efficient alternative.
- **Watchpoints:**
    - Hardware support is limited (e.g., x86-64 supports 4 hardware watchpoints).
    - Software watchpoints are much slower.
- **Conditional breakpoints:**
    - Only break when a condition is true (e.g., `b foo if x == 42`).
- **Macros and scripting:**
    - Automate repetitive tasks with GDB's macro language or Python scripting.
- **Remote debugging:**
    - Debug programs running on another machine (e.g., embedded systems) via network or serial port.

7.6 Print Command Usage

- Print variables, expressions, or call functions:

```
p a + b
p my_struct.member
p my_function()
p exit(1)  // Dangerous: will cause program to terminate
```

7.7 Common GDB Pitfalls and Solutions

Pitfall	Solution/Workaround
Optimized code hard to debug	Compile with `-O0 -g3`
Variables "optimized out"	Use less optimization, check symbol table
Stepping skips lines	Use `stepi` for instruction-level control
Breakpoints not hit	Check for inlined/optimized code
Watchpoints not triggering	Use hardware watchpoints, minimize scope

---

Summary:

- Use static and dynamic tools together for best coverage.
- Defensive programming and rigorous testing are key to robust code.
- Debugging is a process: observe, reproduce, minimize, hypothesize, test, fix, and retest.
- GDB is powerful, but should be used thoughtfully—not as a crutch.
- Always automate and document your debugging and testing processes.