FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Debugging C++ Code

Adnan Alhomssi

Datenbanken und Informationssysteme
Prof. Dr. Viktor Leis

Introduction

- developers generally spend more time debugging than writing code
- this is true for all programming languages, but even more so for C++
- debugging is difficult and frustrating
- bugs can manifest in different ways: crash, wrong result, non-deterministic behavior
- the difficulty is often in finding the bug, not fixing it
- there are a number of tools and techniques to make much debugging easier

General Development Hints

- defensive programming: avoid the bug by considering all possible inputs even if you don't expect them to happen
- if you make assumption about the environment/input, make sure they are valid by writting `asserts` in code
- use git and commit often (not just when it is completely finished)
- use "git diff" to see what you have changed recently
- quickly hacking a fix for an error may introduce a new bug

Tip 1: Compile-Time Warnings

- in GCC and clang the `-Wall` flag enables many useful compile-time warnings
- use this flag always during development and fix all of them ASAP
- catches a significant number of trivial (but sometimes hard-to-find) bugs
- learn to distinguish between compiler warnings, compiler errors, linker errors, and runtime errors

Tip 2: Learn How to Use a Debugger (e.g., gdb)

- compile with `-O0 -g`
- `gdb --args ./a.out ...`
- C-c: interrupt program
- `run`: run the program
- "graphical mode": C-x C-a
- `backtrace`: show backtrace
- `up/down`: up/down a stack frame
- `print`: output variable
- `next`: step one line
- `step`: step into function
- `until`: step over function
- `breakpoint`: set breakpoint
- `continue`: run until next breakpoint
- cout/printf debugging is fine too (sometimes better than stepping)

Tip 2: Learn How to Use a Debugger (e.g., gdb)

- gdb can watch a memory location and breaks once the programm reads from/write to it
- can be helpful to locate the source of memory corruption
- on x86, it can watch max 4 locations (2,4, 8 byte words) using hardware
- otherwise it will fallback to software watchpoints which are very slow
- IDE with GUI usually run gdb in the background and connects to it using TCP
- getting used to gdb on console pays off quickly especially when you have to debug on a remote machine

Tip 3: Undefined Behavior

- in C++ many operations have undefined behavior (UB)
- example: signed integer overflow
- a compiler may do anything on encountering UB
- this is used to optimize code
- one consequence: program may work with `-O0` but not with `-O3`
- compiling with `-fsanitize=undefined` will instrument with UB checks, reporting a warning at runtime when UB is encountered
- it's important to fix all UB, even when they do not result in a bug (a compiler upgrade may add a new optimization)

Tip 4: Memory Issues

- invalid pointers can cause a segmentation fault (SEGFAULT)
- the OS detected an invalid memory access and killed the program
- using a debugger, it is sometimes easy to find out what went wrong
- however, sometimes the memory corruption happened much earlier
- in such cases compiling with `-fsanitize=address` may help (must also link with asan)
- it will instrument every memory access with bounds checks
- this is a major game changer and effectively makes C++ a checked language like Java
- address sanitizer also detect memory leaks

Tip 5: Assertions

- often the consequences of a bug occur much later than when the error actually happened, making it hard to find the root cause
- one way to get closer to the bug is to add `assert(condition)` code during development
- when writing code, one often has certain assumptions (e.g., about function arguments)
- it is a very good idea to state then using asserts
- when an assertion is validated the program stops immediately, which makes it easier to find the root cause of the problem
- if one wants to avoid the performance overhead, one can compile with `-DNDEBUG` to get rid of all assertions
- therefore, one should never have any semantically-meaningful side effects in assertions

Tip 6: Program Localization

- write more tests
- write explicit code checking your data structures
- try to make the test triggering the bug smaller (e.g., reduce data size)
- rule out approach: e.g., if a multi-threaded crashes, check if it crashes single-threaded too
- if the bug occurred recently, check your commit history (`git bisect` allows binary search on commit history)
- google error message (stackoverflow often helps)

Tip 7: Sanity Checks

- sometimes, after a long debugging session, the observed results stop making sense
- Are you editing the right file?
- Are you running the intended binary?
- add a syntax error and check that compilation fails
- add a print statement somewhere in your program and check that you see the output
- take a break, and think about it from a different perspective

Advanced Techniques

- reverse debugging using `rr`, simplifies backtracking
- nice list of techniques: `http://www.cs.cornell.edu/courses/cs312/2006fa/lectures/lec26.html`

Summary

1. use git
2. compile with debug sybmols and warning `-g -Wall`
3. test different optimizaiton levels `-O0 or -O3`
4. use `assert` for your assumptions and invariants
5. step through code with debugger
6. try `-fsanitize=undefined` and `-fsanitize=address`