Snapshot Analysis for C++

a road towards automating C++ debugging

Henning Meyer

October 23, 2025

Slides

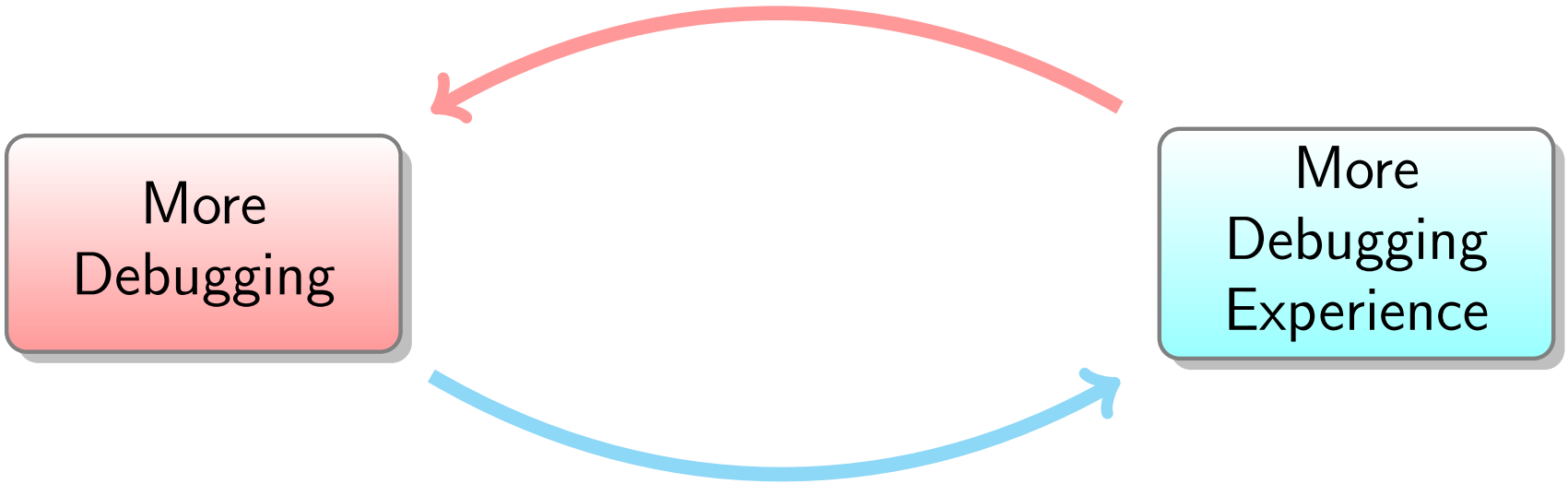
These slides are available at
`https://github.com/core-explorer/blog/blob/main/heap-snapshot.pdf`
The prototype is available at
`https://github.com/core-explorer/core-explorer`

Developer's Journey

- At my first job we had a C++ codebase that was very hard to debug
- it required use of obscure GDB commands (e.g. non-stop mode) and would not work with the debugger integrated into an IDE
- The company hired scientists straight from the university with no prior industry experience
- Lots of race conditions
- We treated debugging as a research problem

Developer's Journey

- At my last job, debugging was treated as a routine problem
- Developers have been working on the same project for ten years
- I had slightly more experience in debugging hard to debug problems
- I get asked to help other teams debugging



- I ended up spending all my time debugging

Cursed Knowledge

Debug Falacy

"Everything will be easier once I get better at debugging"

Outcome

Spend less time on each individual problem, spend more time total by working on (not solving) more problems

Print Falacy

"Everything will be easier once I get better at fixing printer (software) problems"

Outcome

Spend less time on each individual problem, spend more time total by working on (not solving) more problems

Spend Less Time Debugging

The goal is not to get better at debugging,
the goal is spend less time debugging.
The easiest way by far is to aim for higher code quality:

- Write more tests.
- Do more code reviews.
- Use sanitizers.
- Use static analyzers.

Two Types of debugging

- Routine Problems: these problems can be solved by non-experts using tools they are familiar with (their IDE)
- Research problems: these problems require expert knowledge, may require non-standard tool use, may require weeks of investigation

Routine debugging

There are two aspects that contribute to making a problem amenable to routine debugging

- Easy to reproduce: The problem reproduces every time and instantly.
- Easy to analyze: there is a stack trace that points to to the cause of the problem, source code is available. It is a failed assertion, a null pointer dereference or an integer divide by zero.

Easy Reproducibility

The problem reproduces often enough:

- even when changing from a Release build to a Debug build,
- even when changing from normal execution to execution in a debugger
- even when inserting print statements and rebuilding

These conditions rarely hold when the cause is uninitialized memory, a race condition or memory corruption

Research debugging

- Hard or impossible to reproduce: a race condition, an unknown trigger, unavailable hardware or software
- The more customers you have, the more bugs they find that you didn't
- Weak or no connection between cause and effect: memory corruption or memory leaks

Non-Local Effects

C and C++ allow for memory corruption and memory leaks.

- Hard to debug defects with non-local effects.
- The point in time and in space where you notice the effect is not necessarily related to the cause.
- We have very good tools for detecting the presence of a memory leak but very poor tooling for finding the cause.

Two Types of debugging

- The goal is not to get better at debugging
- The goal is to spend less time debugging
- Some problems are research problems and can only be solved by debugging experts
- Turning every developer into an debugging expert is not desirable for businesses or individual developers
- Better tools turn research problems into routine problems

Snapshot Analysis

Snapshot Analysis means data-mining a snapshot of program state (either a core dump or a stopped program in a debugger) for as much information as possible.

- One use case is to aid in finding the root cause for memory corruption.
- Another use case is better understanding of memory consumption.

Debugging as a Research Problem

- binaries are library of functions
- debug information is a library of types
- a core dump is a library of objects
- only local and global variables are easily accessible
- memory on the heap can only be inspected with a hex viewer

Debugging as a Research Problem, automated

- binaries are database of functions (`readelf`, `gdb/lldb`)
- debug information is a database of types (`dwarfdump`, `gdb/lldb`)
- a core dump is a database of objects
- we lack a reliable solution to enumerate objects on the heap

Debugging C and C++ is hard for humans and debuggers

C

- the only way to find objects on the heap is via pointers
- pointers may not need to point at valid objects
- there is no way to tell whether it is safe to dereference a pointer

C++

- memory on the heap is managed via containers and smart pointers
- they are implemented via raw pointers and provide iterators
- iterators may not need to point at valid objects
- there is no way to tell whether it is safe to dereference an iterator
- references must refer to a valid object

Debugging Modern C++ is viable

- C++ containers maintain their invariants by hiding their implementation
- We cannot tell whether an iterator is valid...
- but we can get iterators guaranteed to be valid by calling `begin()` and comparing to `end()`
- dereferencing a valid iterator gives us a valid pointer

Object Discovery Algorithm

1. ignore all existing pointers and iterators
2. find all containers
3. execute `begin()` and `end()` in a VM (e.g. an `x86-64` VM) to get iterators
4. execute `operator*()` in a VM to get a valid pointer
5. execute `operator++()` in a VM to get more iterators
6. This relies on the public interface and works for user-defined container types without requiring additional configuration

Object Discovery Uses

- We can then use our knowledge about containers to check the validity of existing iterators
- We can flag the use of invalidated iterators as an error

Object Discovery Uses

Example

```
example.cpp:10  std::vector<int> vec = {3, 2, 1};
example.cpp:11  auto iterator = vec.begin();
example.cpp:12  vec.push_back(0);
example.cpp:13  vec.erase(iterator);
```

Executing line 13 will cause undefined behavior, even in a hardened implementation. A smart debugger can diagnose this **before** executing the line. A smart time-travel debugger can travel backwards from a crash to this line.

Object Discovery beyond Containers

1. smart pointers work just the same: use `operator bool()` and `operator*()`
2. smart pointers might be declared with an interface, not the concrete type
3. `optional` and `expected` work just the same: `has_value()` and `value()`
4. `std::function` does not have a public interface to get the contained value
5. `std::variant` and `std::any` may or may not have instantiated methods to get the value of a particular type

Automated Time-Travel Debugging

The recordings powering the replay functionality of time travel debuggers can be seen as a timeseries of core dumps.

If we can write a function that analyzes a coredump for memory corruption or memory leaks,
the debugger can travel backwards to first time the problem occurs,
without the need for human (or LLM) guidance.

Automated Memory Leak Detection

- Normal tools test for memory leaks at program exit.
- If we can get a list of all allocations, we can detect *unreachable* allocations.
- Unreachable allocations will lead to memory leaks at program exit, because they cannot be freed.

Automated Allocation Detection

- It is not required that a malloc implementation keeps enough metadata around to recover all allocations.
- A fast malloc implementation will recycle freed allocations to reduce the number of syscalls.
- That requires storing the size of allocated memory, because that information is not provided by arguments to `free()`.

Automated Allocation Detection: GNU libc

- The default malloc implementation on Linux is glibc malloc
- glibc keeps enough metadata to recover the size and address of all allocatons.
- If malloc metadata has been corrupted, the recovery algorithm can detect the corruption.
- This analysis needs to be repeated for other malloc implementations.
- The metadata of malloc implementations is well studied, because exploit developers use corruption of malloc metadata to turn out-of-bounds writes into arbitrary writes.

Causes of Memory Leaks

- An allocation becomes unreachable when the last reachable pointer to it gets overwritten or becomes unavailable.
- This is the cause for the memory leak unless it is a cycle of reference-counted objects.
- We can detect the existence of the cycle.
- A time travelling debugger can travel back in time to the first formation of the cycle.
- If any object involved in the cycle has a vtable pointer, we can use it to recover type information of one and possibly all objects involved in the cycle (RTTI not required)

Diagnosis

1. We have static analyzers for source code.
2. We have static analyzers for binaries.
3. We lack static analyzers for core dumps.
4. We lack analyzers for record-and-replay recordings.