GWDG

Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

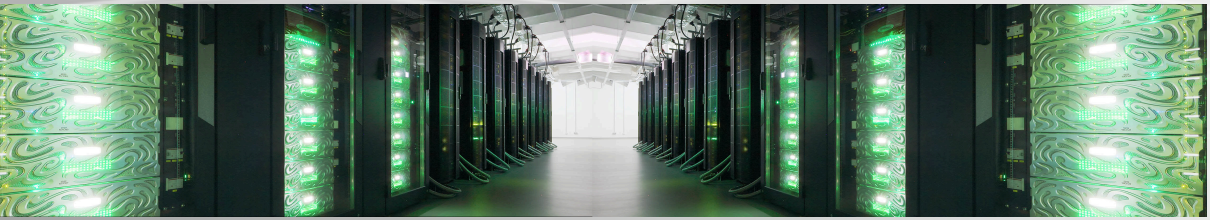
GWDG

https://hpc.gwdg.de

Debugging with GDB, Valgrind, and Sanitiziers

How to make C/C++ debugging bearable

Lars Quentin



02.04.2026

PCHPC

The Scientific Method of Debugging

Steps

1. **Reproduce:**
   - Find bad inputs
2. **Isolate:**
   - Create **MINIMAL** version
3. **Hypothesize:**
   - Try to think why it could happen
4. **Test:**
   - Validate or discard afterwards

The Scientific Method of Debugging

Steps

1. **Reproduce:**
   - Find bad inputs
2. **Isolate:**
   - Create **MINIMAL** version
3. **Hypothesize:**
   - Try to think why it could happen
4. **Test:**
   - Validate or discard afterwards

General Advice

- **Be Radical:**
  - Build an absolutely minimal version
- **No really...**
  - Can you go smaller?
- **Don’t assume anything:**
  - (See JL-example)
- **Use all tools you can get:**
  - GCC is your friend
- **Disable all optimizations**

Outline

1 Intro

2 GCC

3 GDB

4 Sanitizers

5 Valgrind

6 Conclusion

Offloading work to the Compiler

1. Enable debug symbols:
   - `-g`: variable names, line numbers, ...
   - `-g3`: More information, such as macro defs (native DWARF)
   - `-ggdb3`: If you know you use gdb

Offloading work to the Compiler

1. Enable debug symbols:
   - `-g`: variable names, line numbers, ...
   - `-g3`: More information, such as macro defs (native DWARF)
   - `-ggdb3`: If you know you use gdb
2. Disable optimization
   - `-O0` if you want to be safe
   - `-Og` if you trust gcc (I don't)

Offloading work to the Compiler

1. Enable debug symbols:
   - `-g`: variable names, line numbers, ...
   - `-g3`: More information, such as macro defs (native DWARF)
   - `-ggdb3`: If you know you use gdb
2. Disable optimization
   - `-O0` if you want to be safe
   - `-Og` if you trust gcc (I don't)
3. Enjoy modern language: `-std=...`

Offloading work to the Compiler

1. Enable debug symbols:
   - `-g`: variable names, line numbers, ...
   - `-g3`: More information, such as macro defs (native DWARF)
   - `-ggdb3`: If you know you use gdb
2. Disable optimization
   - `-O0` if you want to be safe
   - `-Og` if you trust gcc (I don't)
3. Enjoy modern language: `-std=...`
4. Enable STL checks (more later)

Offloading work to the Compiler

1. Enable debug symbols:
   - `-g`: variable names, line numbers, ...
   - `-g3`: More information, such as macro defs (native DWARF)
   - `-ggdb3`: If you know you use gdb
2. Disable optimization
   - `-O0` if you want to be safe
   - `-Og` if you trust gcc (I don't)
3. Enjoy modern language: `-std=...`
4. Enable STL checks (more later)
5. Enable all warnings:

Offloading work to the Compiler

1. Enable debug symbols:
   - `-g`: variable names, line numbers, ...
   - `-g3`: More information, such as macro defs (native DWARF)
   - `-ggdb3`: If you know you use gdb
2. Disable optimization
   - `-O0` if you want to be safe
   - `-Og` if you trust gcc (I don't)
3. Enjoy modern language: `-std=...`
4. Enable STL checks (more later)
5. Enable all warnings:
   - Just `-Wall`... right?

Enable all relevant warnings

- `-Wall`: Enables “all” common warnings

Enable all relevant warnings

- `-Wall`: Enables “all” common warnings
- `-Wextra`: Even more warnings! ...but now we have all right?

Enable all relevant warnings

- `-Wall`: Enables “all” common warnings
- `-Wextra`: Even more warnings! ...but now we have all right?
- `-Wpedantic`: ISO compliance, relevant for `-O2` Heisenbugs

Enable all relevant warnings

- `-Wall`: Enables “all” common warnings
- `-Wextra`: Even more warnings! ...but now we have all right?
- `-Wpedantic`: ISO compliance, relevant for `-O2` Heisenbugs
- `-Wshadow`: Local variable overshadows another (local, global) variable

Enable all relevant warnings

- `-Wall`: Enables “all” common warnings
- `-Wextra`: Even more warnings! ...but now we have all right?
- `-Wpedantic`: ISO compliance, relevant for `-O2` Heisenbugs
- `-Wshadow`: Local variable overshadows another (local, global) variable
- `-Wconversion`: Warns against conversions that might alter an value

Enable all relevant warnings

- `-Wall`: Enables “all” common warnings
- `-Wextra`: Even more warnings! ...but now we have all right?
- `-Wpedantic`: ISO compliance, relevant for `-O2` Heisenbugs
- `-Wshadow`: Local variable overshadows another (local, global) variable
- `-Wconversion`: Warns against conversions that might alter an value
- `-Wcast-align`: “Warn whenever a pointer is cast such that the required alignment of the target is increased. For example, warn if a `char *` is cast to an `int *` on machines where integers can only be accessed at two- or four-byte boundaries. ”

Enable all relevant warnings

- `-Wall`: Enables “all” common warnings
- `-Wextra`: Even more warnings! ...but now we have all right?
- `-Wpedantic`: ISO compliance, relevant for `-O2` Heisenbugs
- `-Wshadow`: Local variable overshadows another (local, global) variable
- `-Wconversion`: Warns against conversions that might alter an value
- `-Wcast-align`: “Warn whenever a pointer is cast such that the required alignment of the target is increased. For example, warn if a `char *` is cast to an `int *` on machines where integers can only be accessed at two- or four-byte boundaries. ”

Notes

- `-Werror` is good for dev, but leave it off for others please.
- See `https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html`

Asserts

- Remember: Asserts are **zero-cost in release mode**!
- Toggled via `NDEBUG`
- Use many asserts whereever you want ⇒ Never too many
- Assert the impossible (negative) not the expected (positive)
- Best Practise: No side effects!
  - **Bad:** `assert(p = malloc(10));`
  - **Good:** `void *p = malloc(10); assert(p != NULL);`
- Compile time checks via `static_assert`
  - `static_assert(sizeof(int) >= 4, "Weird embedded system");`

STL guarantees

Two possibilities

1. `-D_GLIBCXX_ASSERTIONS`
   - Enables out of bounds checks
   - Actually overwrites `operator[]`
   - No risk in using it
2. `-D_GLIBCXX_DEBUG`
   - Enables out of bounds **and iterator checks**
   - **CHANGES ABI**
     - All your dependencies (Boost etc) also need `-D_GLIBCXX_DEBUG`
   - You need to know what you're doing

Pro-Tipp: `printf`-style debugging

Simplest implementation

```
1  template<class T>
2  void dbg_simple(T t) {
3  #ifndef NDEBUG
4    std::cerr << "[Debug] " << t << std::endl;
5  #endif
6  }
7
```

Pro-Tipp: `printf`-style debugging

Zero cost printf macros

```cpp
#ifndef NDEBUG
void dbg(const auto &val,
    const std::source_location loc = std::source_location::current()) {

  // Get time struct
  using namespace std::chrono;
  auto t{system_clock::to_time_t(system_clock::now())};
  struct tm tb; localtime_r(&t, &tb);

  // Print out
  std::cerr << "[" << std::put_time(&tb, "%H:%M:%S")
  << "] " << loc.function_name() << ":" << loc.line() << " | " << val << '\n';
}
#else
constexpr void dbg(const auto &) noexcept {}
#endif
```

Pro-Tipp: `printf`-style debugging

C version: Only works for `char *`

```c
#ifndef NDEBUG
#include <stdio.h>
#include <time.h>

#define dbg_str(val) do { \
  time_t _t = time(NULL); \
  struct tm _tb; \
  localtime_r(&_t, &_tb); \
  fprintf(stderr, "[%02d:%02d:%02d] %s:%d | %s\n", \
      _tb.tm_hour, _tb.tm_min, _tb.tm_sec, __func__, __LINE__, val); \
} while(0)
#else
#define dbg_str(val) ((void)0)
#endif
```

Outline

1 Intro

2 GCC

3 GDB

4 Sanitizers

5 Valgrind

6 Conclusion

GDB: The GNU Debugger

What is it?

- The de-facto debugger on Linux
- Made for C/C++, can do a lot mroe
- **Advantages over `printf`:**
  - Look into crashes (segfaults)
  - Step Over/Into/Out
  - Call Stack
  - Non-Invasive

Rules for good use

1. Compile with `-ggdb3 -O0`
2. Compile without sanitizers
3. Realize Heisenbugs can exist
   - They obviously change timings
4. As Han Solo said:
   “Good luck... you’re gonna need it”

HELP I HAVE A SEGFAULT (80% of all use cases)

What to do

1. Deep breath
2. Run the program inside gdb
3. Let it crash (SIGSEGV)
4. Look at the *backtrace* how you got there
5. Look at the local state to find the condition
6. If unclear: Real debugging begins

Livedemo: HELP I HAVE A SEGFAULT (80% of all use cases)

What to do

```
# Compile at least unoptimized with debug symbols
gcc -O0 -ggdb3 ./my_program.c -o my_program
gdb ./my_program

(gdb) run <args> # Run the program (with optional args)
# wait for SIGSEGV...
(gdb) bt # Print the call stack
(gdb) bt full # Print the call stack WITH local variables
```

GDB TUI Mode

- GDB has a bad rep because people think its cryptic
- But: It has a Text User Interface (TUI) mode!

How to use it

- **At start:** `gdb -tui ./my_program`
- **On the fly:** Press `Ctrl-X` followed by a

Note: TUIs and `stdout`

- **Problem:** `stdout` can break TUI mode
- **Solution:** Force redraw via `Ctrl-L`

GDB: Most important commands for stepping

- `break main.cc:42` (Set a breakpoint)
- `next` (`n`) (Step over functions)
- `step` (`s`) (Step into functions)
- `finish` (`fin`) (Step out of functions)
- `advance main.cc:42` (Continue until you are there)
- `print var` (`p`) (Show value of variable)
- `display var` (Always show value of variable)

Life Hack: gdb-dashboard

- Python based gdb config with dashboard capabilities
- Looks good, nooby friendly
- Single new command: dashboard
- Even syntax highlighting! (with pysegments)
- github.com/cyrus-and/gdb-dashboard

Outline

1 Intro

2 GCC

3 GDB

4 Sanitizers

5 Valgrind

6 Conclusion

What are Sanitizers?

- Compiler features that add extra checks (instrumentation) at compile time
  - Runtime analysis, no static analysis!
  - Very fast compared to valgrind (ca 2x slowdown)
- Originally developed by Google for LLVM/Clang
  - Also ported to GCC later
- Compile rules:
  - `-g3` for debug symbols
  - `-O0` for easier debugging
  - **`-fno-omit-frame-pointer`** for error traces!
- **Recommendation:** (ASan + UBSan) when developing

AddressSanitizer (ASan) and LeakSanitizer (LSan)

- **Compiler flag:** `-fsanitize=address`
- **What it does:** Tracks allocations and puts redzones around variables to detect illegal access
- **Examples:**
  - Heap out-of-bounds (r/w after `malloc`-region)
  - Stack out-of-bounds (r/w past a local array)
  - Use-after-free
  - Double-free
- **LSan included:** LeakSanitizer is automatically enabled as well.
  - Checks at exit if everything was freed (or thus leaked)

ASAN (LSan) Live Example (Who can see the error)

A good C++ compile command

```c
#include<stdlib.h>
#include<stdio.h>

int main() {
    size_t N=10;
    printf("Allocating array of %zu integers...\n", N);
    int *arr = malloc(N * sizeof(int));

    for (int i = 0; i <= N; ++i)
        arr[i] = i * 2;

    for (int i = 0; i <= N; ++i)
        printf("%d ", arr[i]);
    printf("\n");
}
```

UndefinedBehaviorSanitizer (UBSan)

- **Compiler flag:** `-fsanitize=undefined`
- **What it does:** Detects undefined behavior at RUNTIME (i.e. input dependent)
- **Why its needed:** Compilers are allowed to assume no UB happens
  - Otherwise: You *might* lose correctness (and might is even worse to debug!)
- **Examples:**
  - Signed integer overflow
  - Division by zero
  - Unaligned pointer aliasing
  - Shifting out of bounds (`1u >> 32`)

Other sanitizers (use carefully)

ThreadSanitizer (TSan)

- `-fsanitize=thread`
- Concurrent data accesses without locking
- Deadlocks
- Problems:
  - Up to 10x slower
  - Cannot be combined with ASan+LSan+UBSan

Other sanitizers (use carefully)

ThreadSanitizer (TSan)

- `-fsanitize=thread`
- Concurrent data accesses without locking
- Deadlocks
- Problems:
  - Up to 10x slower
  - Cannot be combined with ASan+LSan+UBSan

MemorySanitizer (MSan)

- `-fsanitize=memory`
- Reading uninitialized variables
- if-branching based on it
- **PROBLEMS:**
  - Requires ALL linked libraries to be recompiled.
    Including `libc++`, your MPI, ...
  - Very easy to get wrong, especially on clusters.
- If you really need it: Try a static analyzer first...

Outline

1 Intro

2 GCC

3 GDB

4 Sanitizers

5 Valgrind

6 Conclusion

Valgrind

What is Valgrind

- Dynamic analysis tool
- Spiritual predecessor to sanitizers
- Synthetic CPU that takes *any* compiled binary, adds instrumentation, and runs it

Valgrind tools (selection)

- **Memcheck:** Memory error detector
- **Cachegrind:** Cache and branch-prediction profiler
- **Callgrind:** Call-grapph generating cache profiler
  - **KCachegrind** GUI for analysis

Valgrind Memcheck

- Memcheck is the default tool in Valgrind.
- It detects memory-management problems
- Roughly equivalent to ASan+LSan+**MSan**
  - Works without recompilation! (opposed to MSan)
- Can detect if your program
  - Leaks memory (with stack traces where it was allocated)
  - Missuses uninitialized values
  - Accesses memory incorrectly (such as alingment or out of bounds)
  - Does illegal frees of heap blocks (double free, stack frees, free on new allocs)
  - Does illegal reads and writes (for example an overlapping `memcpy`)
- **NOTE:** Can be flaky with high optimization levels.

Comparison Memcheck Sanitizers

- Advantages Valgrind
  - No recompilation required
  - Good MSan alternative
  - Allows blackbox debugging (into any third party libraries)
  - No compiler support required
- Disadvantages Valgrind
  - **Very** slow (20-30x overhead, order of magnitude higher than sanitizers)
  - High memory overhead (since it tracks everything)
  - Can get confused when doing vectorization (`-O3`)
- Personal recommendation:
  Use Sanitizers when dev, Valgrind when you have to (you'll know)

Memcheck: Example

Use of uninitialized values in syscalls

```
$ cat > uninit_syscall.c <<'EOF'
#include <stdlib.h>
#include <unistd.h>
int main( void )
{
  char* arr  = malloc(10);
  int*  arr2 = malloc(sizeof(int));
  write( 1 /* stdout */, arr, 10 );
  exit(arr2[0]);
}
EOF
```

Memcheck: Example

Use of uninitialized values in syscalls

```
$ gcc -o uninit_syscall uninit_syscall.c -Wall -g
$ module load valgrind
$ valgrind ./uninstall_syscall
...
Syscall param write(buf) points to uninitialised byte(s)
    at 0x25A48723: __write_nocancel (in /lib/tls/libc-2.3.3.so)
    by 0x259AFAD3: __libc_start_main (in /lib/tls/libc-2.3.3.so)
Address 0x25AB8028 is 0 bytes inside a block of size 10 alloc'd
    at 0x259852B0: malloc (vg_replace_malloc.c:130)
    by 0x80483F1: main (a.c:5)
Syscall param exit(error_code) contains uninitialised byte(s)
    ...
```

Outline

1 Intro

2 GCC

3 GDB

4 Sanitizers

5 Valgrind

6 Conclusion

How to compile safely (For Sanitizers/General)

A good C++ compile command

```
g++ -Wall -Wextra -Wpedantic \
    -Wshadow -Wconversion -Wcast-align \
    -ggdb3 -O0 -std=c++23 \
    -fsanitize=address -fsanitize=undefined \
    -fno-omit-frame-pointer \
    -D_GLIBCXX_ASSERTIONS \
    ./src/main.cc -o main
```

How to compile safely (For gdb/valgrind)

A good C++ compile command

```
# NO SANITIZERS
g++ -Wall -Wextra -Wpedantic \
    -Wshadow -Wconversion -Wcast-align \
    -ggdb3 -O0 -std=c++23 \
    -fno-omit-frame-pointer \
    -D_GLIBCXX_ASSERTIONS \
    ./src/main.cc -o main
```

Conclusion: What to use when

- **Logic Errors:** GDB or `printf`-debugging
- **Crashes (segfaults):** GDB or ASAN+UBSan
- **Memleaks:** Valgrind or LSan
- **Uninitialized Memory:** Valgrind
- **Data Races/Unintentional Indeterministim:** TSan
- **Problem with closed-source lib:** Valgrind