Back to the Basics: Sequential Debugging

Jonathan Schmalfuß

Chair of Scientific Computing
University of Bayreuth

March 18, 2025

Sequential debugging

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.” - Brian Kernighan, professor at Princeton University.

Helpful compilation flags: g++ compiler

Warning flags

- `-Wall`: Enables many warning flags
- `-Werror`: Converts any warning into an error
- `-Wextra`/`-W`: Additional warnings not in `-Wall`
- `-pedantic`/`-Wpedantic`: ISO standard compliance warnings
- Specific warnings:
  - `-Wconversion`, `-Wcast-align`, `-Wunused`, `-Wshadow`, `-Wold-style-cast`
  - `-Wpointer-arith`, `-Wcast-qual`, `-Wmissing-prototypes`, `-Wno-missing-braces`
- the more the better, but only if you resolve them - Recommended Flags
- use at least: `-Wall -Wextra -Wshadow -Wnon-virtual-dtor -pedantic`

CPP/C++ Compiler Flags and Options

Trivial example: -Wall

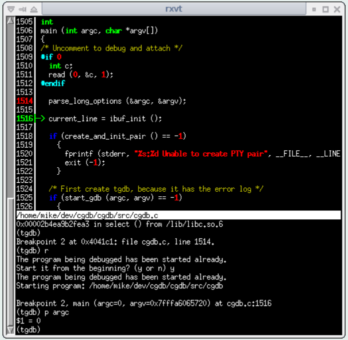
```cpp
#include <iostream>

int main() {
    int x; // Uninitialized variable warning
    std::cout << "Uninitialized value of x: " << x << "\n";

    return 0;
}
```

```
$ g++ wall_example.cpp
$ g++ -Wall -Werror wall_example.cpp
wall_example.cpp: In function 'int main()':
wall_example.cpp:5:55: error: 'x' is used uninitialized
5 |     std::cout << "Uninitialized value of x: " << x << "\n";
  |                                                       ^~~~
cc1plus: all warnings being treated as errors
```

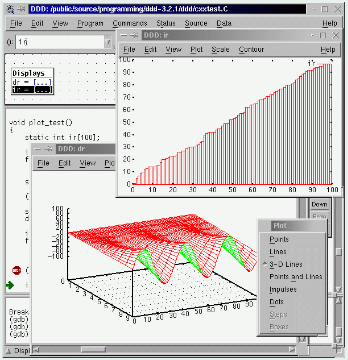
Debugger Frontends

- lightweight terminal based cgdb
- old school DDD
- python based - browser-based gdbgui
- modern interpretation seer

- common wisdom: learn one, then others are easier to learn



```
1505  int
1506  main (int argc, char *argv[])
1507  {
1508  /* Uncomment to debug and attach */
1509  #if 0
1510    int c;
1511    read (0, &c, 1);
1512  #endif
1513
1514    parse_long_options (&argc, &argv);
1515
1516->  current_line = ibuf_init ();
1517
1518    if (create_and_init_pair () == -1)
1519      {
1520        fprintf (stderr, "%s:%d Unable to create PTY pair", __FILE__, __LINE
1521        exit (-1);
1522      }
1523
1524    /* First create tgdb, because it has the error log */
1525    if (start_gdb (argc, argv) == -1)
1526      {
/home/mike/dev/cgdb/cgdb/src/cgdb.c
0x00002b4ea9b2fea3 in select () from /lib/libc.so.6
(tgdb)
Breakpoint 2 at 0x4041c1: file cgdb.c, line 1514.
(tgdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
The program being debugged has been started already.
Starting program: /home/mike/dev/cgdb/cgdb/src/cgdb

Breakpoint 2, main (argc=0, argv=0x7fffa6065720) at cgdb.c:1516
(tgdb) p argc
$1 = 0
(tgdb)
```



```
void plot_test()
{
    static int ir[100];
```

```
Break
(gdb)
(gdb)
(gdb)
```

All hail the king: GNU command line debugger: gdb

```cpp
#include<iostream>

long factorial(int n)
{
    long result(1);
    while(n>0)
    {
        result*=n;
        n--;
    }
    return result;
}
```

```cpp
int main()
{
    int n(0);
    std::cout << "Input n to n!: ";
    std::cin >> n;
    long val=factorial(n);
    std::cout << "Input n = " << n;
    std::cout << " , n! = " << val << "\n";
    return 0;
}
```

Live session inspired by hackingcpp: Debugging With gdb

GNU command line debugger: gdb

key / command	short form	meaning
`<Enter>`		repeat previous command
`<Tab>`		complete command or function name
`run [<arg>...]`	`r [<arg>...]`	run program (with command line argument(s))
`break <loc>`	`b <loc>`	set breakpoint at beginning of function or at a specific line
`step`	`s`	next instruction, step into function
`next`	`n`	next instruction, step over function
`jump <loc>`	`j <loc>`	jump to location (useful for exiting long/endless loops)
`continue`	`c`	continue until next breakpoint or end of program
`until <loc>`	`u <loc>`	continue until location (function /line)
`finish`	`fin`	finish (step out of) current function
`print <expression>`	`p`	print value of expression, e.g., variable
`info breakpoints`	`i b`	list all breakpoints
`info locals`	`i locals`	list local variables and their values
`backtrace`	`bt`	show call stack

Table: Useful gdb Commands

Valgrind’s tool suite

- Memcheck: detects memory-management problems:

  `valgrind -leak-check=full -show-leak-kinds=all -track-origins=yes -verbose <myexecutable>`

- example usage

```
$ g++ -g Wall -Werror out_of_bounds.cpp -o out_of_bounds.out
$ valgrind ./out_of_bounds.out
==182905== Memcheck, a memory error detector
==182905== Copyright (C) 2002-2022, and GNU GPL'd, by Seward et al.
==182905== Using Valgrind-3.21.0 and LibVEX;
==182905== Command: ./a.out
==182905==
==182905== Invalid read of size 8
...
```

Sanitizers

Sanitizers

- AddressSanitizer - memory error detector: `-fsanitize=address`[1]
- UndefinedBehaviorSanitizer - undefined behavior detector: `-fsanitize=undefined`
- ThreadSanitizer - data race detector: `-fsanitize=thread`

[1] GCC compiler flags for fsanitize

Example: Sanitizers

```
$ g++ -g -fsanitize=address segfault.cpp -o a_segfault.out
$ ./a_segfault.out
AddressSanitizer:DEADLYSIGNAL
=================================================================
==188435==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000
...

==188435==ABORTING
```

Honorable Mentions: Static analyzers / Linters

- cppcheck -to detect bugs and undefined behavior and dangerous coding constructs
- Clang-Tidy - diagnosing and fixing typical programming errors
- include-what-you-use - confirms your includes are sufficient

Time travel debugging

Idea: run gdb; stop at the point where problem occurs; walk backwards figure out what happend
⇒ theoretically supported by gdb via commands such as
`reverse-next`

- Problem: avx instruction ⇒ unusable most of the time
- Solution: `rr`[a]: record a failure once, then debug the recording, deterministically, as many times as you want

[a] https://rr-project.org/

Now what?

Use the tools / test them on simple and more complex examples

In the corresponding github project work through the folder `02_sequential_programs` content. The available tools on the PC-Pool workstations are gdb / sanitizer / valgrind.

Possible tasks:

- How many bugs can be found using only the warning flags? Is there a special type that is difficult to be found using warning flags?
- Learn to read the sanitizer / valgrind output. Attempt the trivial examples first, then try the more difficult ones.
- Hints / Solutions are available for the more complicated one, use them!
- Last but not least: Ask Questions, not only to me but also to each other about your understanding.