Praktikum C-Programmierung 2026

Debugging und Profiling

Jannek Squar

2026-05-20

Scientific Computing
Universität Hamburg

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Outline

- Motivation
- Debugging
  - Beispiel
  - Verwendung von GDB
- Valgrind
  - Memcheck
  - Callgrind
  - Massif

Table of Contents

Motivation

Debugging

- Beispiel
- Verwendung von GDB

Valgrind

- Memcheck
- Callgrind
- Massif

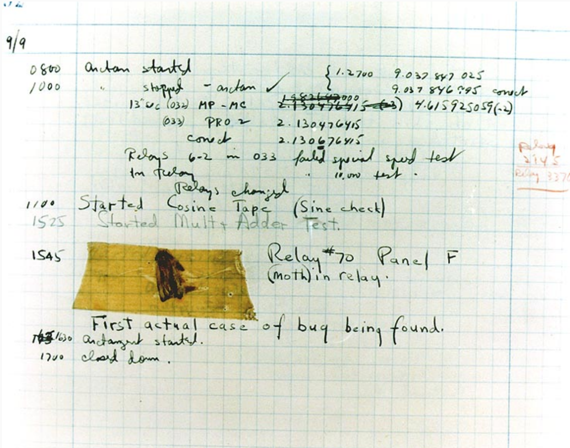


Abbildung 1: Dokumentation eines “echten” Computer-Bugs [1]

Bugs

- Debugging im HPC-Umfeld potentiell schwierig:
  - Komplexes Zusammenspiel: Rechenknoten, Netzwerk, Speicher, Software-Stack, etc.
  - Nichtdeterminismus (auch in sequentiellen Programmen möglich)
  - Mehr Hardware -> geringere MTBF (Mean Time Between Failures)
- Bug-Sorten:
  - Syntax-Error
  - Runtime-Error
- Bugfreiheit ist algorithmisch unentscheidbar
  - Problematisch, wenn Programm nicht abstürzt

Table of Contents

Motivation

Debugging

- Beispiel
- Verwendung von GDB

Valgrind

- Memcheck
- Callgrind
- Massif

```c
#include <stdio.h>

int testFunc(int i) {
  if (i > 0) {
    return testFunc(i - 1) + i;
  } else if (i == 0) {
    return 0;
  }
}
```

```c
int main(void) {
  int ergebnis, eingabe;

  eingabe = 10;
  ergebnis = testFunc(eingabe);
  printf("Ausgabe: %d\n", ergebnis);

  eingabe = -10;
  ergebnis = testFunc(eingabe);
  printf("Ausgabe: %d\n", ergebnis);

  return 0;
}
```

```
$ gcc -o recursion.x recursion.c;
$ ./recursion.x
Ausgabe: 55
Ausgabe: -10
```

recursion.c

```c
#include <stdio.h>

int testFunc(int i) {
  if (i > 0) {
    return testFunc(i - 1) + i;
  } else if (i == 0) {
    return 0;
  }
}
```

```c
int main(void) {
  int ergebnis, eingabe;

  eingabe = 10;
  ergebnis = testFunc(eingabe);
  printf("Ausgabe: %d\n", ergebnis);

  eingabe = -10;
  ergebnis = testFunc(eingabe);
  printf("Ausgabe: %d\n", ergebnis);

  return 0;
}
```

```
$ gcc -o recursion.x recursion.c;
$ ./recursion.x
Ausgabe: 55
Ausgabe: -10
```

recursion.c

```c
#include <stdio.h>

int testFunc(int i) {
  if (i > 0) {
    return testFunc(i - 1) + i;
  } else if (i == 0) {
    return 0;
  }
}
```

```c
int main(void) {
  int ergebnis, eingabe;

  eingabe = 10;
  ergebnis = testFunc(eingabe);
  printf("Ausgabe: %d\n", ergebnis);

  eingabe = -10;
  ergebnis = testFunc(eingabe);
  printf("Ausgabe: %d\n", ergebnis);

  return 0;
}
```

```
$ gcc -o recursion.x recursion.c;
$ ./recursion.x
Ausgabe: 55
Ausgabe: -10
```

Erste Maßnahmen - Beobachtungen

```
$ ./recursion.x
[...]
$ echo $?
0
```

- Programm stürzt nicht ab

```
$ gcc -o recursion.x recursion.c -O0; ./recursion.x
Ausgabe: 55
Ausgabe: -10
$ gcc -o recursion.x recursion.c -O3; ./recursion.x
Ausgabe: 55
Ausgabe: 0
```

- Unterschiedliche Optimierungslevel verändern Ausgabe → UB
- `printf` an strategischen Punkten einfügen → irgendwann unübersichtlich

Erste Maßnahmen

```
$ gcc -o recursion.x recursion.c -Wall
recursion.c: In function ‘testFunc’:
recursion.c:9:1: warning: control reaches end of non-void function [-Wreturn-type]
    9 | }
      | ^
```

- Compiler-Flags (hier: `-Wall`) können wichtige Hinweise geben
- Unterstützung durch Tools:
  - Valgrind
  - GDB (GNU Debugger)
    - erlaubt deterministischen, kleinschrittigen Code-Durchlauf
    - Open Source Command line Debugger
    - Backend für andere Debugger

Erste Schritte mit GDB

- Kompilieren mit `-Og -g` oder `-Og -ggdb`
- Interessante GDB-Flags:
  - Übergabe von Parametern: `--args ./app arg1 ... argN`
  - Ordner mit Sourcen: `--directory=DIR`
  - Weitere Literatur: `gdb --help` und `man gdb`
- Neukompilierung während Debugging möglich

How to GDB (I)

Ausführung

- **r**un
- **c**ontinue
- **n**ext
- **s**tep
- **fi**nish
- **q**uit/**k**ill

Ausgabe

- **p**rint [/f] [var]
- ptype var
- display [/f] var
- undisplay var
- tui enable/disable
- **l**ist [n|+|-|.|n,m]

How to GDB (II)

Break-/Watch-Points

- **b**reak [file:]line | [file:]func [if condition]
- **tb**reak ...
- info b
- disable/enable/delete n
- clear [[file:]line|[file:]func]

An laufendes Programm anhängen: `gdb --pid PID`

Programm-Stack

- **back**t**race [n]
- frame [n]
- up/down [n]

Live-Demo

Demo

`recursion.c`

Programmstart, Grundlagen, Navigation durch Backtrace

How to GDB (III)

- set var=value
- print *pointer
- print *dynArray@LENGTH
- info threads
- thread n
- help CMD

- GDB-Frontends [4]
- Online-Variante [3]
- GDB Quick Reference [2]
- Blog [5]

Live-Demo

Demo

`memory.c`

Ausgabe von statischen und dynamisch allokiertem Speicher

Table of Contents

Motivation

Debugging

- Beispiel
- Verwendung von GDB

Valgrind

- Memcheck
- Callgrind
- Massif

Was ist ... ? [6]

- *"Valgrind is a dynamic binary instrumentation (DBI) framework"*
- Valgrind core + tool plug-in = Valgrind tool

  Core: Instrumentierung der Client-Applikation

  Plug-In: Durchführung der Analyse
- Einfache Bedienung
  - Sourcecode unverändert
  - Robust und weit verbreitet
  - Vielseitige Kontrollmöglichkeiten
- Open Source (GNU General Public License, version 2)

Funktionsweise [6]

- Dynamic Binary Instrumentation $\rightarrow$ Arbeit auf Maschinencode
  - Kein Neukompilieren notwendig
  - Kein Zugriff auf Source-Code notwendig
- Längere Laufzeit durch Emulation
  - Simuliert *Gast*-CPU
  - Shadow Values ("Software-Wrapper") [7]
  - Kernel bleibt Black-Box $\rightarrow$ Systemcalls teuer
  - Serialisiert Threads

  $\Rightarrow$ meist schlechtere Laufzeit um mindestens eine Größenordnung

Valgrind-Tools

- **Memcheck** (Test auf Speicherfehler)
- Cachegrind (Statistik über Cache-Nutzung)
- **Callgrind** (Erstellt Call-Graph)
- **Massif** (Statistik über Heap-Nutzung)
- Helgrind (Test auf Race Conditions)
- DRD (Test auf Fehler beim Multithreading)

Verwendung

Kompilieren:

```
$ gcc -g -o app app.c
```

Ausführen:

```
$ valgrind --tool=<tool> [valgrind-options] ./app [app-options]
```

Hinweis

- Kompilieren mit `-g` Flag nicht vergessen!
- Hilfe: `man valgrind` oder `valgrind --help`

Motivation aus *Einführung Heap*

Fehlerhafte Speicherverwendung:

- Speicherlecks
- Zugriff auf nicht reservierten Speicher
- Nutzung nach `free()`
- Freigabe von nicht reserviertem Speicher
- Mehrfacher Aufruf von `free()`
- Verlorene Pointer

Außerdem:

- Verwendung nicht-initialisierter Werte
- ...

Motivation: Memcheck [10]

- Default-Tool von Valgrind
- Manuelle Verwaltung von dynamischen Speicher in C
  - Vorteil: User kann Speicher-Zugriffe optimieren
  - Nachteil: User kann sich das Leben beliebig schwer machen

$\Rightarrow$ Memcheck prüft Programm auf fehlerhafte Speicher-Nutzung

```
$ valgrind ./app [app-options]
```

Demo

`heap.c`

Speicherfehler sichtbar machen und mit `valgrind` eingrenzen

Motivation: Callgrind [9]

- Visualisierung des Programmablaufs
- Erkennung potentieller Bottlenecks
- Genauer als einfaches Sampling
  - Aber: Verändertes Timing durch Instrumentierung
- Relativ großer Overhead zur Laufzeit kurzer Methoden
- Ergebnisse in `callgrind.out.<pid>`
- Programme zur Auswertung
  - Texteditor (bitte nicht!)
  - `callgrind_annotate`
  - `kcachegrind`

```
$ valgrind --tool=callgrind ./app [app-options]
```

Beispiel: callgrind_anotate auf callgraph.c

```
4 ( 0.00%)  size_t myAllocation(int **myOutput) {
1 ( 0.00%)    size_t const mySize = 10;
12 ( 0.01%)    *myOutput = (int *)malloc(mySize * sizeof(int));
734 ( 0.48%)  => /usr/src/debug/glibc-2.42-13.fc43.x86_64/malloc/malloc.c:malloc (1x)
675 ( 0.44%)  =>
↪  /usr/src/debug/glibc-2.42-13.fc43.x86_64/elf/../sysdeps/x86_64/dl-trampoline.h:_dl_runtime_resolve_xsave
↪  (1x)
.
45 ( 0.03%)    for (size_t i = 0; i < mySize; ++i) {
70 ( 0.05%)      myOutput[0][i] = (int)i;
.              }
1 ( 0.00%)    return mySize;
2 ( 0.00%)  }
.
40 ( 0.03%)  void myPrint(int myValue) {
64 ( 0.04%)    printf("%d\n", myValue);
8,869 ( 5.85%)  => /usr/src/debug/glibc-2.42-13.fc43.x86_64/stdio-common/printf.c:printf (10x)
643 ( 0.42%)  =>
↪  /usr/src/debug/glibc-2.42-13.fc43.x86_64/elf/../sysdeps/x86_64/dl-trampoline.h:_dl_runtime_resolve_xsave
↪  (1x)
30 ( 0.02%)  }
.
5 ( 0.00%)  int main(int argc, char const *argv[]) {
.              int *myOutput;
4 ( 0.00%)    size_t const mySize = myAllocation(&myOutput);
1,544 ( 1.02%)  => callgraph.c:myAllocation (1x)
```

Beispiel: kcachegrind auf callgraph.c

File View Go Settings Help

Open... Back Forward Up Relative Cycle Detection Relative to Parent Shorten Templates Instruction Fetch

Flat Profile

Search: Search Query (No Grouping)

Incl.	Self	Called	Function	Location
100.00	0.01	(0)	0x0000000000002110	ld-2.30.so
62.25	0.51	1	_dl_start	ld-2.30.so
61.74	0.31	1	_dl_sysdep_start	ld-2.30.so
46.73	0.56	1	dl_main	ld-2.30.so
37.60	13.33	4	_dl_relocate_object	ld-2.30.so
36.72	0.01	1	_start	03_callgraph-example.x
36.72	0.03	1	(below main)	libc-2.30.so
35.79	0.06	1	main	03_callgraph-example.x: 03_callgraph-example.c
32.59	0.06	3	_dl_runtime_resolve_xsave	ld-2.30.so
31.48	0.06	1	myAllocation	03_callgraph-example.x: 03_callgraph-example.c
31.13	0.01	1	malloc_hook_ini	libc-2.30.so
30.80	0.04	1	ptmalloc_init.part.0	libc-2.30.so
30.35	30.33	1	_dl_addr	libc-2.30.so
24.66	9.22	101	_dl_lookup_symbol_x	ld-2.30.so
15.44	11.23	101	do_lookup_x	ld-2.30.so
14.58	14.58	1	__GI___tunables_init	ld-2.30.so
4.71	0.00	3	0x0000000004002090	(unknown)
4.71	0.04	3	_dl_catch_exception	ld-2.30.so
4.60	0.21	3	_dl_map_object	ld-2.30.so
4.26	0.06	10	myPrint	03_callgraph-example.x: 03_callgraph-example.c
4.22	2.32	92	check_match	ld-2.30.so
4.13	0.43	1	_dl_map_object_deps	ld-2.30.so
3.78	1.46	10	__vfprintf_internal	libc-2.30.so
3.41	0.02	2	openaux	ld-2.30.so
3.37	0.14	9	printf	libc-2.30.so
2.43	2.43	153	strcmp	ld-2.30.so
1.90	1.05	2	_dl_map_object_from_fd	ld-2.30.so
1.77	0.77	30	_IO_file_xsputn@@GLIBC_2...	libc-2.30.so
1.41	0.03	1	handle_preload_list	ld-2.30.so
1.27	0.01	1	do_preload	ld-2.30.so
1.25	0.00	1	0x00000000040020f0	(unknown)
1.25	0.01	1	_dl_catch_error	ld-2.30.so
1.22	0.01	1	map_doit	ld-2.30.so
1.15	0.01	1	_dl_receive_error	ld-2.30.so
1.14	0.00	1	version_check_doit	ld-2.30.so
1.14	0.03	1	_dl_check_all_versions	ld-2.30.so
1.11	0.45	1	open_path	ld-2.30.so
1.10	0.70	4	_dl_check_map_versions	ld-2.30.so
1.09	0.19	1	_dl_init_paths	ld-2.30.so
1.01	0.04	1	_dl_init	ld-2.30.so
0.98	0.08	4	call_init.part.0	ld-2.30.so
0.86	0.19	15	_IO_file_overflow@@GLIBC...	libc-2.30.so
0.83	0.00	1	exit	libc-2.30.so
0.82	0.05	1	__run_exit_handlers	libc-2.30.so
0.82	0.10	3	_dl_fixup	ld-2.30.so
0.78	0.11	1	init_cacheinfo	libc-2.30.so
0.74	0.54	18	open_verify.constprop.0	ld-2.30.so
0.67	0.08	3	handle_intel.constprop.0	libc-2.30.so
0.61	0.13	1	_dl_fini	ld-2.30.so
0.61	0.15	1	_dl_load_cache_lookup	ld-2.30.so
0.59	0.59	6	intel_check_word.isra.0	libc-2.30.so

main

Types Callers All Callers Callee Map Source Code

#	Ir	Source
0		--- From '/home/dreistein/Git/02_WR/c-praktikum/2019-2020/slides/06_debugging_and_valgrind/code/03_callgraph-example.c' ---
25		}
26		
27		int main(int argc, char const *argv[])
28	0.00	{
29		int *myOutput;
30	0.00	size_t const mySize = myAllocation(myOutput);
	31.48	1 call to 'myAllocation' (03_callgraph-example.x: 03_callgraph-example.c)
31		
32	0.02	for(size_t i = 0; i < mySize; i++)
33		{
34	0.03	myPrint(myOutput[i]);
	4.26	10 calls to 'myPrint' (03_callgraph-example.x: 03_callgraph-example.c)
35		}
36		
37	0.00	return 0;
38	0.00	}

_start 35.79 %
1 x
(below main) 35.79 %
1 x
main 35.79 %
1 x
10 x
myAllocation 31.48 %
myPrint 4.26 %
1 x
9 x
_dl_runtime_resolve_xsave 31.42 %
printf 3.37 %
10 x
__vfprintf_internal

Parts Callees Call Graph All Callees Caller Map Machine Code

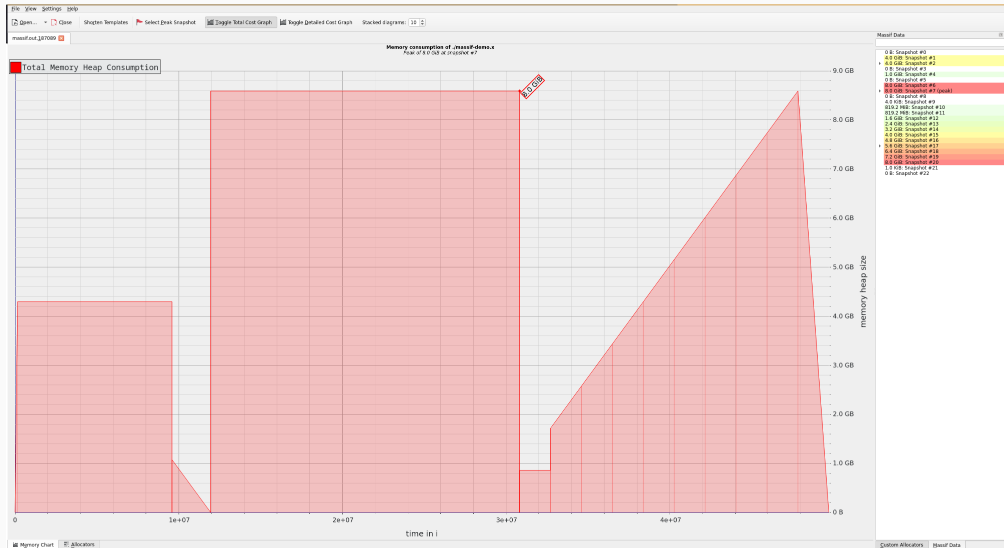
callgrind.out.22380 [1] - Total Instruction Fetch Cost: 220 114

Motivation: Massif [8]

- Misst Speicherverbrauch
  - „Wirklich verwendeter“ Heap-Speicher
  - Indirekt verwendeter Heap-Speicher (Metadaten, alignment, usw.)
  - Stack (default: off, weil langsamer)
- Ergebnisse in `massif.out.<pid>`
- Rückgriff auf gdb notwendig bei oom
- Programme zur Auswertung:
  - `ms_print`
  - `massif-visualizer`

```
$ valgrind --tool=massif [--time-unit=B] ./app [app-options]
```

Motivation: Massif



References

[1] Wikimedia Commons: *The First "Computer Bug"*. 1947.
`https://commons.wikimedia.org/wiki/File:H96566k.jpg`

[2] Roland H. Pesch: *GDB QUICK REFERENCE*. 2025.
`https://sourceware.org/gdb/onlinedocs/refcard.pdf`

[3] Mritunjay Singh Sengar: *OnlineGDB - online compiler and debugger for c/c++*. 2018.
`https://www.onlinegdb.com`

[4] Marina Kalashina: *GDB front ends and other tools*. 2018.
`https://sourceware.org/gdb/wiki/GDB%20Front%20Ends`

References ...

[5] Kevin Pouget: *How Does a C Debugger Work? (GDB Ptrace/x86 example)*. 2014. `https://blog.0x972.info/?d=2014/11/13/10/40/50-how-does-a-debugger-work`

[6] Nicholas Nethercote and Julian Seward: *Valgrind*. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI '07). ACM Press, 2007. doi: `https://doi.org/10.1145/1250734.1250746`

[7] Nicholas Nethercote and Julian Seward: *How to shadow every byte of memory used by a program*. In: Proceedings of the 3rd international conference on Virtual execution environments (VEE '07). ACM Press, 2007. doi: `https://doi.org/10.1145/1254810.1254820`

References ...

[8] Valgrind User Manual: *Massif: a heap profiler.*
`http://valgrind.org/docs/manual/ms-manual.html`

[9] Valgrind User Manual: *Callgrind: a call-graph generating cache and branch prediction profiler.*
`http://valgrind.org/docs/manual/cl-manual.html`

[10] Valgrind User Manual: *Memcheck: a memory error detector.*
`http://valgrind.org/docs/manual/mc-manual.html`