

# Debugging mit GDB

Albrecht Oster

Proseminar „C - Grundlagen und Konzepte“

## Inhaltsverzeichnis

<b>1</b>	<b>Was ist Debugging?</b>	<b>2</b>
1.1	Fallbeispiel	3
1.2	Mögliche Vorgehensweise zur Ursachenfindung	4
<b>2</b>	<b>Was ist GDB?</b>	<b>5</b>
2.1	Befehlsübersicht	5
2.2	Anwendung auf Fallbeispiel	6
2.3	Stackframes	7
<b>3</b>	<b>GDB in Entwicklungsumgebungen</b>	<b>8</b>

# 1 Was ist Debugging?

Die Entwicklung einer Software lässt sich in mehrere Arbeitsschritte unterteilen. Neben dem eigentlichen Schreiben des Programmcodes ist unter anderem auch die Arbeit des Debugging von besonderer Bedeutung. Hierbei wird die Software - oder auch nur ein Teil davon - auf Herz und Nieren geprüft. Sollte es zu unerwartetem Verhalten kommen, ist es wichtig, die Ursache dieses Problems zu finden und anschließend zu beheben. Der Fokus liegt hierbei auf der Ursachenforschung, da diese in den meisten Fällen die schwierigere Aufgabe ist. Hat man den Programmfehler erkannt, ist der Weg zur Behebung meist nicht mehr weit.

Um den Aufwand möglichst gering zu halten, ist es wichtig, möglichst früh mit dem Debugging zu beginnen. So sollten selbst kleinere Teile des Codes in frühem Stadium der Entwicklung bereits getestet werden, da das Programm mit der Zeit komplexer und somit auch die Fehleruntersuchung schwieriger wird. Daher ist es üblich, dass die beiden Schritte des Entwickelns und Debuggings zeitlich parallel verlaufen.

Fehler, die im Debugging gefunden werden können, müssen sich nicht nur auf eindeutige Programmabstürze beschränken. Besonders heikel wird es bei Fehlfunktionen, die das Programm nicht abstürzen lassen, sondern in fehlerhaften Ergebnissen resultieren. Es kommt häufig vor, dass diese erst nach intensivem Testen oder gar der Auslieferung der Software an den Kunden entdeckt werden.

Im Folgenden untersuchen wir ein Codebeispiel, das gleich mehrere Fehler aufweist.

## 1.1 Fallbeispiel

Das folgende Codebeispiel soll die Zahlen von 1 bis 10 in ein Array speichern und anschließend in umgekehrter Reihenfolge wieder ausgeben.

```
#include <stdio.h>

void schreibeZahlen(int zahlen[], int anzahl)
{
    for (int i = 0; i < anzahl; i--)
    {
        zahlen[i] = i + 1;
    }
}

void gibZahlAus(int zahlen[], int index)
{
    printf("%d\n", zahlen[index]);
}

void gibZahlenAus(int zahlen[], int anzahl)
{
    for (int i = anzahl; i >= 0; i++)
    {
        gibZahlAus(zahlen, i);
    }
}

int main(int argc, char *argv[])
{
    int zahlen[10];
    schreibeZahlen(zahlen, 10);
    gibZahlenAus(zahlen, 10);
}
```

Tatsächlich jedoch stürzt das kompilierte Programm gleich zu Beginn ab. Der angezeigte Fehler `Segmentation fault` deutet auf einen Speicherzugriffsfehler hin. Mehr Informationen stehen uns zu diesem Zeitpunkt nicht zur Verfügung.

Es wurde somit ein Fehler entdeckt, die Ursache ist jedoch noch unbekannt und muss nun untersucht werden. Da dieser Fehler einen Crash zur Folge hat, ist die Entdeckung trivial, was nicht immer so sein muss.

## 1.2 Mögliche Vorgehensweise zur Ursachenfindung

Man kann sich an den nun gefundenen Fehler ohne weitere Hilfsmittel heranarbeiten. Dies erfolgt durch Bearbeiten des Codes, indem man Debugging-Informationen auf der Konsole ausgeben lässt. Dazu reicht meist ein simpler `printf`-Aufruf gepaart mit hilfreichen Informationen für den Entwickler.

```
printf("Zahlen in Array geschrieben.\n");
```

Diese Zeile fügen wir an einer sinnvollen Stelle, in diesem Fall unter dem Aufruf der Funktion `schreibeZahlen` in `main`, ein. Sollte „Zahlen in Array geschrieben“ nun während der Ausführung des Programms in der Konsole erscheinen, lässt sich daraus schließen, dass der Absturz erst nach dieser Zeile passieren muss und dessen Ursache somit weiter eingekreist werden kann. Da dies in unserem Beispiel nicht der Fall ist und die Zeile nicht erscheint, können wir aus dem Umkehrschluss folgern, dass der Fehler vor dieser Zeile auftreten muss.

Weiterhin kann es nun hilfreich sein, den aktuellen Kontext in die Debugging-Informationen mit einzubeziehen. So könnten wir noch während der Ausführung der Funktion `schreibeZahlen` durch Angabe des jeweils aktuellen Index über den Fortschritt informieren.

```
printf("Schreibe Zahl %d in Index %d\n", i+1, i);
```

Platzieren wir diese Zeile vor der Zuweisung in der `for`-Schleife der Funktion `schreibeZahlen`, erhalten wir jeweils einen Hinweis, welche Zahl an welchem Index abgelegt wird. Führen wir unser Programm mit dieser Änderung nun aus, erfahren wir, dass die erste Zahl an Index 0 gespeichert wird, die zweite bei Index -1, die dritte bei Index -2 und so weiter. Offensichtlich zählt die Schleife rückwärts und iteriert somit über negative Indizes, was sehr schnell zum Zugriff auf ungültige Speicherbereiche und folglich dem Absturz führt. Ein Fehler ist hiermit gefunden und kann nun behoben.

Grundsätzlich ist es immer sinnvoll, Debugging-Informationen zusätzlich zu weiteren Hilfsmitteln wie GDB einzusetzen. Dies hilft besonders bei Fehlern, die während der Ausführung erst nach einer Weile auftreten und die Suche mit Hilfe von GDB sehr mühselig machen würden.

Unser Code kann an dieser Stelle korrigiert werden, indem wir den Schleifen-Zähloperator in `schreibeZahlen` zu `i++` ändern. Es befinden sich noch weitere Fehler in unserem Code, die wir im Folgenden mit GDB untersuchen werden.

## 2 Was ist GDB?

Der GNU Debugger oder kurz GDB kann den Arbeitsschritt des Debuggings enorm vereinfachen und neue Möglichkeiten eröffnen, die die traditionelle Vorgehensweise beim Debugging, wie sie im vorigen Kapitel demonstriert wurde, nicht bietet. Er ist Teil des *GNU-Projekts*, somit Open Source, und kann auf nahezu allen Betriebssystemen und Prozessorarchitekturen sowie einer Vielzahl an Programmiersprachen wie *C*, *C++* oder *Java* angewandt werden.

GDB bietet die Möglichkeit, den Ablauf eines Programms Zeile für Zeile zu verfolgen und sogar in diesen einzugreifen. So kann man das Programm an beliebiger Stelle unterbrechen und sich die Variablenwerte zu diesem Zeitpunkt aus dem Speicher ausgeben lassen. Bereits diese elementaren Funktionen des Debuggers könnten die Schritten aus dem Kapitel zuvor überflüssig machen.

### 2.1 Befehlsübersicht

Zunächst muss der Programmcode mit dem Flag `-g` kompiliert werden, damit die Binärdatei mit Informationen für das Debugging versehen wird. Anschließend lässt sich `gdb` mit Angabe der Binärdatei ausführen. GDB wartet nun auf Instruktionen. Der normale Programmablauf lässt sich mit `run` anstoßen, weitere Befehle sind im Folgenden aufgelistet.

<code>break my_func</code> <code>break example.c:12</code>	Unterbricht den Programmablauf durch einen sogenannten Breakpoint an beliebiger Stelle, die man entweder durch Angabe einer Zeile oder einer Funktion bestimmt. So hält GDB jedes Mal inne, wenn jene Zeile erreicht oder jene Funktion aufgerufen wird.
<code>print my_var</code>	Gibt den aktuellen Wert der genannten Variable aus.
<code>watch my_var</code>	Überwacht die genannte Variable und informiert, sobald sich diese ändert. Ähnliches für Lesezugriffe ist mit <code>rwatch</code> möglich.
<code>continue</code>	Setzt den normalen Programmablauf bis zum nächsten Breakpoint oder dem Ende des Programms fort.
<code>next</code>	Führt eine Instruktion in der aktuellen Funktion aus und unterbricht danach erneut.
<code>step</code>	Wie <code>next</code> , springt jedoch bei Funktionsaufrufen in die aufgerufene Funktion.
<code>finish</code>	Führt den Rest der aktuellen Funktion aus und springt dann aus ihr heraus zurück in die ursprünglich aufrufende Funktion.

## 2.2 Anwendung auf Fallbeispiel

Wir bedienen uns nun also GDB, um die restlichen Fehler und deren Ursachen in diesem Programm zu finden. Zunächst hilft uns eine wesentliche Funktion von GDB, für die wir den Ablauf nur starten müssen: Es zeigt uns, in welcher Zeile das Programm abstürzt. Folgende Ausgabe erhalten wir, indem wir das Programm einfach mit `start` durchlaufen lassen:

```
Program received signal SIGSEGV, Segmentation fault.  
0x0000000100000e8e in gibZahlAus (zahlen=0x7fff5fbff8a0, index=138712) ...  
13 printf("%d\n", zahlen[index]);
```

GDB gibt uns alle Informationen die wir brauchen. So wissen wir jetzt, dass das Programm in der Funktion `gibZahlAus` abstürzt und sehen gleich auch die letzten lokalen Variablenwerte in dieser Funktion. Es wird schnell klar, dass der Index zu groß ist und somit das Problem darstellt.

Versuchen wir dieses Problem zu untersuchen, indem wir einen Breakpoint auf diese Funktion setzen, um sie Zeile für Zeile zu durchlaufen.

```
(gdb) break gibZahlAus  
Breakpoint 2 at 0x100000e86: file example.c, line 13.  
(gdb) start  
Starting program: example  
Breakpoint 1, gibZahlAus (zahlen=0x7fff5fbff8a0, index=10) at example.c:13  
13 printf("%d\n", zahlen[index]);
```

Wir befinden uns jetzt in der Funktion `gibZahlAus` und erkennen bereits anhand der Variablenwerte den Fehler. Der erste Index, der an diese Funktion übergeben wurde, ist 10, kann jedoch höchstens 9 sein, da unser Array nur 10 Elemente hält. Der Fehler steckt in der Funktion `gibZahlenAus`, der Startindex der Schleife muss `anzahl-1` sein. Wir korrigieren dies und führen das Programm erneut aus. Es stürzt immer noch ab, also setzen wir wieder den Breakpoint, um zu schauen, wo dieses Mal das Problem liegt.

```
Breakpoint 2, gibZahlAus (zahlen=0x7fff5fbff8a0, index=9) at example.c:13  
13 printf("%d\n", zahlen[index]);
```

Der Index stimmt nun. Mit `print` können wir uns den eigentlichen Wert des Array an diesem Index ausgeben und können so überprüfen, ob das Array auch zu Beginn richtig gefüllt wurde.

```
(gdb) print zahlen[index]  
$1 = 10
```

Auch dies stimmt. Bis hier hin scheint alles in Ordnung zu sein. Bewegen wir uns nun also Zeile für Zeile mit `next` oder `step` fort, bis uns etwas auffällt.

```
(gdb) next  
Breakpoint 2, gibZahlAus (zahlen=0x7fff5fbff8a0, index=10) at example.c:13
```

Schon nach einem Schritt zeigt sich, dass der Index wieder falsch ist, da er inkrementiert statt dekrementiert wird.

Wir ändern also den Zähleroperator der Schleife in `i--` und erkennen nach erneutem Ausführen, dass das Programm jetzt endlich ordnungsgemäß funktioniert.

## 2.3 Stackframes

Stackframes stellen bei der Ausführung von Programmen eine bedeutende Rolle, da sie den zur Verfügung stehenden Speicher für lokale Variablen eines jeden Unterprogramms abgrenzen sowie für den weiteren Ablauf wichtige Informationen, wie zum Beispiel die Rücksprungadresse, enthalten. In unserem Fall sind die Funktionen Unterprogramme. So hat nicht nur `main` einen Stackframe, sondern auch alle aufgerufenen Funktionen. Wird in `main` eine Funktion aufgerufen, wird für diese ein separater Stackframe erzeugt und wieder aufgelöst, sobald die Funktion verlassen wird. Durch weitere Funktionsaufrufe aus anderen Funktionen heraus oder gerade bei Rekursion kann so eine lange Kette an Stackframes, der Stacktrace, entstehen.

GDB bietet viele Möglichkeiten, Stackframes und den Stacktrace zu untersuchen. Die wichtigsten werden im Folgenden gelistet:

<code>frame</code>	Zeigt Informationen zum aktuellen Stackframe an. So lassen sich dessen Variablen und deren Werte anzeigen.
<code>backtrace</code>	Zeigt die Kette der Stackframes an, über die der aktuelle Stackframe erreicht wurde.
<code>up / down</code>	Mit diesen Befehlen kann man sich innerhalb des Backtrace nach oben oder unten zu anderen Stackframes bewegen, um weitere Untersuchungen durchzuführen.

Wir wenden dies nun auf unser Programm an, indem wir einen Breakpoint auf die Funktion `gibZahlenAus` setzen, ausführen und `frame` eingeben.

```
(gdb) frame
#0 gibZahlenAus (zahlen=0x7fff5fbff8a0, anzahl=10) at example.c:18
```

Wir befinden uns also in Stackframe 0 und uns stehen die lokalen Variablen `zahlen` und `anzahl` zur Verfügung. Ein `backtrace` zeigt uns die Kette, die zu diesem Stackframe führt.

```
(gdb) backtrace
#0 gibZahlenAus (zahlen=0x7fff5fbff8a0, anzahl=10) at example.c:18
#1 0x0000000100000f3e in main (argc=1, argv=0x7fff5fbff908) at example.c:28
```

Die aktuelle Funktion wurde also von der `main`-Funktion aufgerufen. Gehen wir nun einen Schritt weiter, „steppen“ bis in die nächste Funktion (`gibZahlAus`) und lassen uns wieder einen `backtrace` ausgeben.

```
(gdb) backtrace
#0 gibZahlAus (zahlen=0x7fff5fbff8a0, index=9) at example.c:13
#1 0x0000000100000ee4 in gibZahlenAus (zahlen=0x7fff5fbff8a0, anzahl=10) ...
#2 0x0000000100000f3e in main (argc=1, argv=0x7fff5fbff908) at example.c:28
```

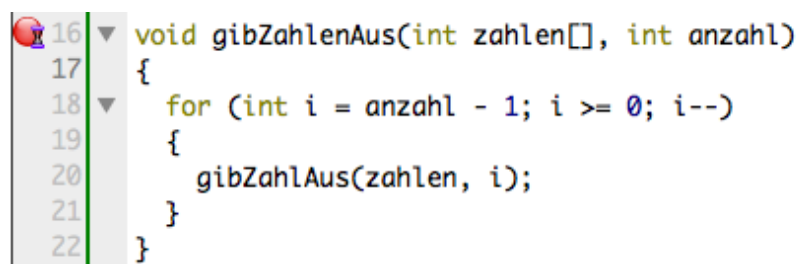
So lässt sich der Programmverlauf leicht ablesen: `gibZahlAus` wurde von `gibZahlenAus` aufgerufen, was wiederum in `main` aufgerufen wurde.

Durch `up` und `down` kann man sich in dieser Kette auf- und abbewegen und damit auch die anderen Stackframes in der Kette untersuchen. Gehen wir also von diesem Punkt aus einen Stackframe nach oben, befinden wir uns in `gibZahlenAus` und können auch dort Variablenwerte ausgeben lassen.

### 3 GDB in Entwicklungsumgebungen

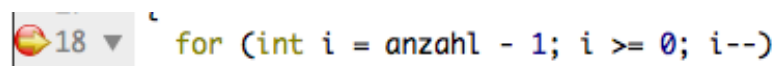
GDB ist von Haus aus ein klassisches Konsolenprogramm, wird jedoch von vielen Entwicklungsumgebungen wie *Eclipse*, *Netbeans* oder *QT Creator* eingebunden, die ihre eigene grafische Oberfläche mitbringen. Alle essentiellen Funktionen von GDB stehen dabei zur Verfügung. Im Folgenden wird dies anhand des *QT Creator* veranschaulicht.

Das Setzen von Breakpoints ist in grafischen Oberflächen besonders intuitiv, da es nur eines Klicks neben der entsprechenden Zeile bedarf. Damit wird die Zeile optisch durch einen roten Punkt markiert und später beim Debugging automatisch als Breakpoint gesetzt. Wir wenden dies auf die Funktion `gibZahlenAus` an.



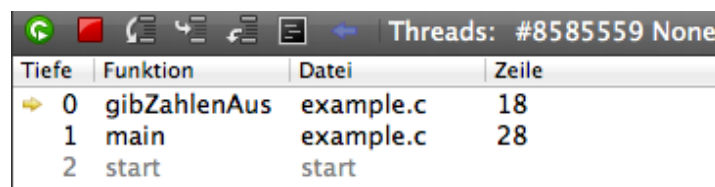
```
16 void gibZahlenAus(int zahlen[], int anzahl)
17 {
18     for (int i = anzahl - 1; i >= 0; i--)
19     {
20         gibZahlAus(zahlen, i);
21     }
22 }
```

Starten wir erneut das Debugging. Das Programm hält nun bei der markierten Zeile inne, markiert diese mit einem gelben Pfeil, um den aktuellen Standpunkt hervorzuheben.



```
18 for (int i = anzahl - 1; i >= 0; i--)
```

Gleichzeitig sehen wir im unteren Panel von *QT Creator* neben der Ausgabe des Programms jederzeit den Stacktrace, der der Konsolendarstellung ähnelt.



Tiefe	Funktion	Datei	Zeile
0	<b>gibZahlenAus</b>	example.c	18
1	main	example.c	28
2	start	start	

Auch hier ist der aktuelle Stackframe wieder mit einem gelben Pfeil markiert. Statt `up` und `down` klickt man einfach den gewünschten Stackframe an, um in diesen zu wechseln.

Alle Befehle zum weiteren Durchlaufen des Programms finden sich in der grauen Zeile darüber. Der grüne Button ist äquivalent zu `continue`, der rote Button bricht das Debugging ab und die



drei Pfeil-Buttons dahinter bedeuten `next`, `step` und `finish`. Klicken wir also auf den ersten Pfeil-Button springen wir in die nächste Zeile, während der gelbe Pfeile mitwandert.

```
→ 20      gibZahlAus(zahlen, i);
```

Ein Klick auf den zweiten Pfeil-Button lässt uns in die aufgerufene `gibZahlAus` springen.

```
11 void gibZahlAus(int zahlen[], int index)
12 {
13     printf("%d\n", zahlen[index]);
14 }
```

Natürlich bietet uns die Oberfläche auch Einblick in unsere lokalen Variablen, die im linken Panel aufgelistet sind.

Name	Wert	Typ
▶ [statics]		
index	9	int
zahlen	1	int

Moderne Entwicklungsumgebungen bieten - wie GDB selbst - noch viele weitere Möglichkeiten des Debuggings, die Grundlagen haben wir hiermit in einem groben Überblick kennengelernt.

# Literatur

- Wikipedia: „GNU Debugger“  
[https://en.wikipedia.org/wiki/GNU\\_Debugger](https://en.wikipedia.org/wiki/GNU_Debugger)
- „GDB: The GNU Project Debugger“  
<http://www.gnu.org/software/gdb/documentation/>
- Samuel Huang - „GDB Tutorial“  
<http://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf>
- O'Reilly - „Debuggen mit gdb“  
<http://www.oreilly.de/german/freebooks/rlinux3ger/ch142.html>
- „Quick Guide to Gdb“  
<http://condor.depaul.edu/glancast/373class/docs/gdb.html>