

# Rechnerorganisation

## Hilfreiche Werkzeuge

### v1.1



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Paul Adelmann  
16. Mai 2020

---

## Inhaltsverzeichnis

---

<b>1</b>	<b>Einführung</b>	<b>2</b>
<b>2</b>	<b>GNU-Debugger</b>	<b>2</b>
2.1	Der GDB . . . . .	2
2.2	Installation . . . . .	2
2.3	Usecase . . . . .	2
2.4	Erste Schritte . . . . .	3
2.5	GDB sinnvoll nutzen . . . . .	3
2.6	Hinweise . . . . .	5
<b>3</b>	<b>Disassembler - objdump</b>	<b>6</b>
3.1	Beispielprogramm . . . . .	6
3.2	objdump nutzen . . . . .	6
3.3	Ausgabe analysieren . . . . .	6
3.4	Hinweis: grep . . . . .	7

---

## 1 Einführung

---

Im Modul Rechnerorganisation beinhalten viele der Übungen praktische Aufgaben, welche es erfordern, zu Programmieren oder in anderen Weisen mit Code umzugehen. In diesem Dokument werden daher verschiedene Tools kurz vorgestellt und/oder ausführlich erläutert, welche die Dinge im RO-Alltag erleichtern können.

---

## 2 GNU-Debugger

---

---

### 2.1 Der GDB

---

GDB ist die Abkürzung für den GNU Debugger, ein C (und C++) Debugger, welcher dem GNU Projekt entsprungen ist. Ein **Debugger** ist ein Werkzeug, welches es erlaubt, ein anderes Programm auf Fehler zu untersuchen, indem beispielsweise die Ausführung an beliebigen vorher markierten Stellen, sogenannten **Haltepunkten** oder **Breakpoints**, gestoppt werden kann, genauere Angaben über Fehler und sogar einzelne Variablenwerte während der Ausführung ausgegeben werden können.

In diesem Abschnitt wird der generelle Umgang mit gdb anhand eines einfachen Beispiels erklärt, die wichtigsten Befehle werden hervorgehoben.

---

### 2.2 Installation

---

Für die allgemeine Bearbeitung der Aufgaben empfiehlt es sich, den clientssh-arm zu nutzen, auf diesem ist gdb bereits vorinstalliert. Ein Tutorial zur Einrichtung und Nutzung ist im moodle Kurs zu finden.

Der gdb kann aber selbstverständlich auch auf dem Privatrechner genutzt werden. Unter (GNU/)Linux lässt sich das passende package über so gut wie alle pagemanager finden, beispielsweise kann es in Debian-basierenden Systemen wie Ubuntu oder Mint mittels apt installiert werden:

```
$ sudo apt-get update
$ sudo apt-get install gdb
```

Die Installation von gdb unter Windows/ OSX ist etwas komplizierter und würde den Rahmen hier etwas sprengen, ich rate daher lieber dazu, den clientssh zu nutzen. Tutorials können jedoch über eine online Suche gefunden werden. An dieser Stelle sei aber auch darauf hingewiesen, dass manche IDEs wie Eclipse teilweise den gdb direkt beiläufig installieren, womit Debugging und die Installation auch sehr angenehm möglich sind.

---

### 2.3 Usecase

---

Um die Funktionsweise von gdb zu verstehen, beginnen wir mit einem primitiven Beispielprogramm in C, welches nicht so funktioniert, wie der Programmierer erwartet hatte. Das gewünschte Verhalten ist an der Seite kommentiert.

```
1 #include <stdio.h>
2
3 double sampleMath(){    // Beispielberechnung
4     int a = 40;
5     int b = 4;
6
7     a = a / b;           // a = 10
8     double c = a / b;    // c = 2.5
9     double d = c * 2;    // d = 5
10    return d;
11 }
12
13 int main (){
14     double d = sampleMath(); // Berechne Wert
15     printf("%d\n", d); // Gebe berechneten Wert aus: 5
16     return 0;
17 }
```

Listing 1: tutorial.c

---

Man könnte erwarten, dass das Programm die Ausgabe 5 liefert, jedoch ist die Ausgabe seltsam, es scheinen mehrere Fehler aufgetreten zu sein.

```
$> gcc tutorial.c -o tutorial
$> ./tutorial
> -390191256
```

#### Listing 2: Ausgabe des Programms

Gerade in solchen Situationen ist es gut, mittels eines Debuggers an das Problem heranzugehen. In jeder Zeile eine Ausgabe der aktuellen Variablen zu implementieren, führt nämlich gerade bei noch größeren Projekten schnell dazu, dass man den Überblick verliert, und das Problem ist dann eventuell trotzdem noch nicht gelöst.

Zeit also, mit dem Debuggen zu beginnen.

---

## 2.4 Erste Schritte

Beim Compilen eines C Programms gehen eine Menge Informationen, wie beispielsweise Funktionsnamen verloren. Um diese zu erhalten, wird das Programm zunächst mit der `-g` Flag compiled

```
$> gcc -g tutorial.c -o tutorial
```

Dann kann gdb mit unserem gerade kompilierten Programm gestartet werden

```
$> gdb tutorial
```

Hinweis am Rande: Um gdb zu stoppen, lautet der Befehl *quit*.

gdb wurde gestartet und weitere Terminaleingabe richten sich erstmal nurnoch an gdb. Zu Beginn bietet es sich an, sich in den Code, den man debuggen möchte überhaupt anzusehen. Dazu kann das Codelayout unseres Programmes im oberen Fenster eines Splitscreens angezeigt werden, sobald dieser läuft. Dazu öffnen wir die Ansicht mittels

```
$(gdb)> layout next
```

Wird nun das Programm gestartet, ist in der oberen Ansicht der Code sichtbar. Zum Starten des Codes nutzen wir

```
$(gdb)> run
```

So weit so gut, unser Programm gibt uns hier nun direkt den fehlerhaften Wert aus und es kommt dazu die Meldung, dass es normal terminiert hat.

Sollte das debuggte Programm beispielsweise in einer Endlosschleife hängen oder lange für eine Berechnung brauchen, kann die Programmausführung jederzeit mit `Strg+C` unterbrochen werden. gdb markiert dabei dann im oberen Fenster, welche Zeile gerade ausgeführt wurde.

---

## 2.5 GDB sinnvoll nutzen

In unserem Beispiel setzen wir nun zuerst einen Breakpoint an der Funktion `sampleMath`, um dann jede Berechnung einzeln überprüfen zu können. Die Ausführung wird dann beim nächsten Start an der Stelle des Breakpoints angehalten, sodass wir den Programmfluss besser steuern können. Um einen Breakpoint an einer Funktion oder Methode zu setzen, lautet der Befehl `break Funktionsname`, alternativ kann aber auch einfach eine Zeilennummer angegeben werden. Die zwei folgenden Befehle führen also zum gleichen Ergebnis:

```
$(gdb)> break sampleMath
$(gdb)> break 3
```

Wird das Programm nun mit `run` gestartet, wird die Zeile des Breakpoints markiert und die Ausführung ist gestoppt. Nun gibt es mehrere Möglichkeiten, den aktuellen Programmzustand zu untersuchen und zu verändern.

Sollte das Layout kaputt gehen, was über die ssh Verbindung gerne mal geschieht, lässt sich das mittels

```
$(gdb)> refresh
```

meist wieder beheben.

Um sehr feinfühlig einen Bug zu untersuchen, bietet es sich an, die Auswirkungen jeder einzelnen Zeile Code, oder eventuell sogar jedes einzelnen Statements zu untersuchen.

Um eine weitere Zeile in der Ausführung weiterzuschreiten, lautet der Befehl `next`, oder kurz `n`.

```
$(gdb)> next
$(gdb)> n
```

Sollte in einer solchen Zeile beispielsweise ein Funktionsaufruf geschehen, wird dieser als Ganzes behandelt. Will man die Ausführung ein wenig feinfühlicher begutachten, gibt es noch den `step` command, welcher nun beispielsweise auch in die Funktion springen würde. Eine gute Faustregel ist dabei, dass `next` in die nächste Zeile des aktuellen Kontextes (also der aktuellen Prozedur) springt und `step` in das nächste Statement, also auch aus dem aktuellen Kontext heraus.

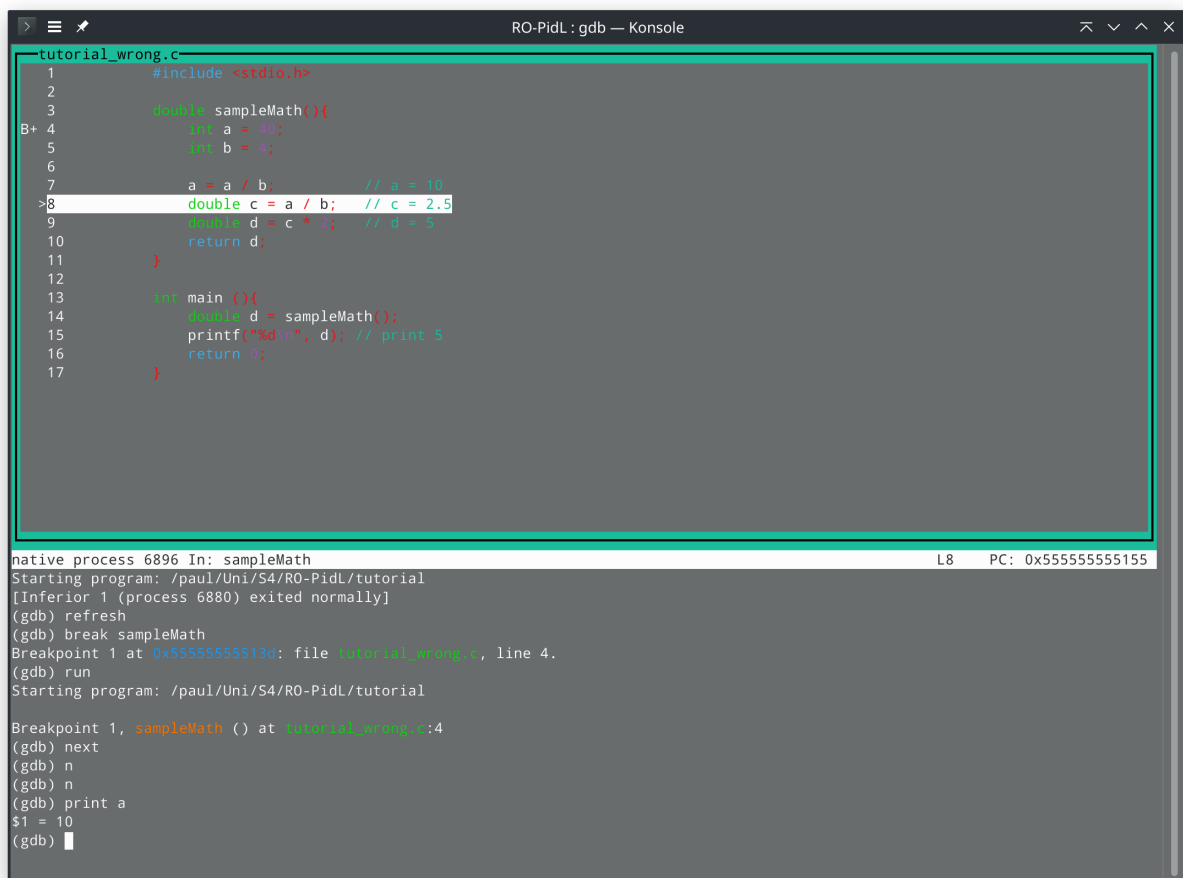
```
$(gdb)> step
```

Mittels dieser Befehle schreiten wir im Programm bis zur ersten interessanten Berechnung fort, natürlich hätte man auch einfach direkt in Zeile 8 einen Breakpoint setzen können. Nun sind wir an einer Stelle angekommen, an der wir den ersten Fehler vermuten. Um eine Variable zu untersuchen, kann man sich mittel `print variable` den Wert dieser ausgeben lassen.

```
$(gdb)> print var
```

Mithilfe dieser Methode stellt sich heraus, dass `c` in Zeile 8 statt 2.5 hier bereits nur den Wert 2 hat, wir haben also bereits den ersten Fehler gefunden. Alle Variablen und weitere Laufzeitparameter lassen sich auch via `info locals` ausgeben, es werden alle Werte des aktuellen Stackframes auf einmal ausgegeben. Ein Array `arr` lässt sich mittels `print *arr@len` komplett ausgeben. Dabei ist `len` die Länge des Arrays, das `*` ist zum dereferenzieren notwendig, ansonsten würden nur die Speicheradressen ausgegeben werden.

Abbildung 1: Debuggen des Programmes mittels gdb



```
tutorial_wrong.c
1  #include <stdio.h>
2
3  double sampleMath(){
4      int a = 40;
5      int b = 4;
6
7      a = a / b; // a = 10
8      double c = a / b; // c = 2.5
9      double d = c * 2; // d = 5
10     return d;
11 }
12
13 int main(){
14     double d = sampleMath();
15     printf("d\n", d); // print 5
16     return 0;
17 }

native process 6896 In: sampleMath
Starting program: /paul/Uni/S4/RO-PidL/tutorial
[Inferior 1 (process 6880) exited normally]
(gdb) refresh
(gdb) break sampleMath
Breakpoint 1 at 0x5555555513d: file tutorial_wrong.c, line 4.
(gdb) run
Starting program: /paul/Uni/S4/RO-PidL/tutorial

Breakpoint 1, sampleMath () at tutorial_wrong.c:4
(gdb) next
(gdb) n
(gdb) n
(gdb) print a
$1 = 10
(gdb)
```

---

Nach einer kurzen Recherche stellt sich heraus, dass in C für eine Fließkommazahlendivision mindestens einer der Operanden bereits ein Fließkommawert sein muss. Im Code könnte man das einfach beheben, indem man in der Zeile einen der Operanden zu einem `double` castet. Wir könnten das Programm hier jetzt also abbrechen, den Fehler beheben, neu compilieren und dann erneut debuggen. Da wir das erwartete Ergebnis aber auch einfach im Kopf berechnet haben und jetzt eigentlich nicht den ganzen Aufwand zur Fehlersuche erneut betreiben wollen, können wir hier einfach schnell den Wert der Variable `c` anpassen. Dazu nutzen wir

```
$(gdb)> set variable var=value
$(gdb)> set variable c=2.5
```

So kann man nun also das gesamte Programm debuggen. Bei größeren Programmen bietet es sich eventuell an, einen neuen Breakpoint zu setzen und den eben gesetzten Breakpoint wieder zu entfernen. Das Setzen eines Breakpoints ist bereits bekannt, allerdings lassen diese sich auch wieder entfernen und man kann eine Übersicht über alle Breakpoints erhalten via

```
$(gdb)> info break
$(gdb)> del 1 // Löscht breakpoint Nummer 1
```

Soll ein breakpoint zwar nicht direkt gelöscht werden, ist aber gerade dennoch nicht benötigt, kann er nach belieben de- und wieder re-aktiviert werden:

```
$(gdb)> en 1 // Breakpoint 1 aktivieren
$(gdb)> dis 1 // Deaktivieren
```

Letztenendes stellen wir fest, dass selbst mit korrektem Rückgabewert der Funktion `sampleMath` die Ausgabe fehlerhaft ist. Da versucht wird, ein `double` auszugeben, muss der `printf` Funktion dementsprechend der korrekte format specifier für einen floating point-Wert übergeben werden, hier also `%f` statt `%d`.

Es empfiehlt sich, für weitere Informationen diesbezüglich in das `printf` Tutorial zu schauen, welches ebenfalls im RO-Moodle Kurs zu finden ist.

Das Debugging stellt sich somit als erfolgreich heraus und alle Fehler konnten behoben werden. Mittels

```
$(gdb)> quit
```

kann `gdb` nun beendet und die gefundenen Fehler anschließend im Quelltext behoben werden.

---

## 2.6 Hinweise

Das Nutzen des `gdb` funktioniert auf dem `clientssh-arm` normalerweise ohne Einschränkungen, sollte `gdb` dort den Fehler `selected architecture not compatible` liefern, wurde der Code wahrscheinlich nicht korrekt mittels `gcc` wie beschrieben compilert. Es sollte also überprüft werden, dass das compilieren korrekt ausgeführt wurde, bei weiteren Problemen kann gerne eine Frage im Moodle Forum gestellt werden.

Das Scrollen in der Code-Ansicht funktioniert mittels der Pfeiltasten, wirkt der Code zuerst abgeschnitten, sollte also überprüft werden, ob er nicht vielleicht einfach aus dem Viewport heraus gescrollt wurde. Über Putty unter Windows kann dies erfahrungsgemäß selten zu Problemen führen, es muss dann sichergestellt werden, dass der Fokus ordentlich im Terminalfenster liegt.

---

### 3 Disassembler - objdump

---

Manchmal ist es sehr hilfreich, sich die Maschinenbefehle oder die Mnemonics eines bereits compilierten Programmes anzusehen. Hat man ein Programm vorliegen, kann bei dieser Aufgabe ein sogenannter Disassembler helfen. Es gibt eine Vielzahl an Tools unterschiedlicher Komplexität, für unsere Anwendung in diesem Kurs reicht es aber erstmal, sich einfach ein wenig mit der Funktion des `objdump` vertraut zu machen.

`Objdump`'s grundlegende Funktion ist es, Informationen über Objektdateien anzuzeigen. Wir nutzen es hier nun um uns anzusehen, wie das folgende einfache C-Programm am Ende übersetzt wurde.

---

#### 3.1 Beispielprogramm

---

Gegeben sei folgendes Beispielprogramm:

```
1 #include <stdio.h>
2
3 int main (){
4     int a = 4;
5     int b = 3;
6     int c = a - b;
7     a = b * 2 + c;
8     printf("%d\n", a);
9     return 0;
10 }
```

Listing 3: `sample.c`

```
$> arm-linux-gnueabi-gcc sample.c -o sample
$> ./sample
7
```

---

#### 3.2 `objdump` nutzen

---

Nun soll `sample` untersucht werden. Dazu wir die `S`-Flag genutzt, um `objdump` eine Quelle mitzugeben:

```
$> objdump -S sample
```

`S` steht dabei für Source und impliziert direkt weitere Flags, sollte weiteres Interesse bestehen, bietet es sich an, in die man `page(s)` zu schauen oder ein wenig in der weiteren [Dokumentation](#) zu lesen.

Die Ausgabe kann im ersten Moment etwas überrumpelnd wirken. Nun wird jedoch ersichtlich, dass beim Compilen (und Linken) im Hintergrund mehr Prozesse ablaufen als man vermuten mag.

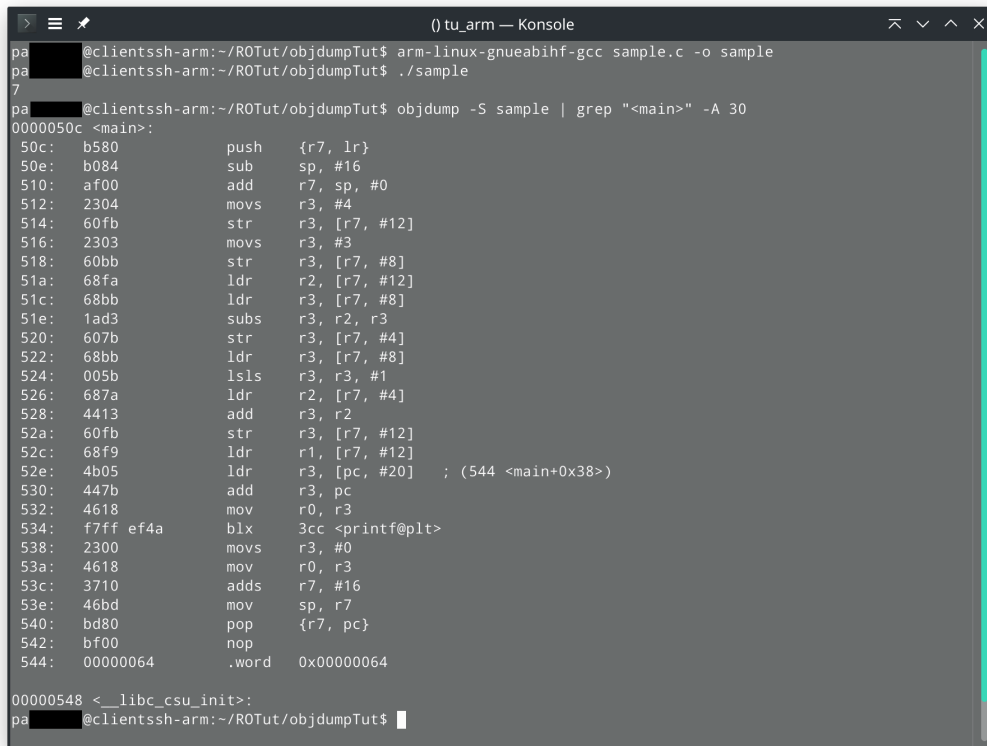
---

#### 3.3 Ausgabe analysieren

---

Bei genauerer Untersuchung stellt man fest, dass die meisten Anweisungen erstmal ignoriert werden können. Die Inhalte sind in Tabellenform dargestellt, für jeden Abschnitt bzw. Marker gibt es einen eigenen Absatz. Ganz links wird die Adresse der Instruktion abgebildet, rechts daneben der Wert an dieser Stelle, also die Instruktion selbst. Dabei ist zu beachten, dass die Werte Hexadezimal kodiert sind. Rechts daneben sind die Befehle in Mnemonics übersetzt, um lesbar zu sein. Da das Programm mit dem `arm-gcc` compiliert wurde, sind diese automatisch in ARM-Assembler angegeben bzw. interpretiert.

Abbildung 2: Untersuchen der objdump Ausgabe



```
() tu_arm — Konsole
pa @clientssh-arm:~/ROTut/objdumpTut$ arm-linux-gnueabi-gcc sample.c -o sample
pa @clientssh-arm:~/ROTut/objdumpTut$ ./sample
7
pa @clientssh-arm:~/ROTut/objdumpTut$ objdump -S sample | grep "<main>" -A 30
0000050c <main>:
50c: b580      push    {r7, lr}
50e: b084      sub     sp, #16
510: af00      add     r7, sp, #0
512: 2304      movs    r3, #4
514: 60fb      str     r3, [r7, #12]
516: 2303      movs    r3, #3
518: 60bb      str     r3, [r7, #8]
51a: 68fa      ldr     r2, [r7, #12]
51c: 68bb      ldr     r3, [r7, #8]
51e: 1ad3      subs    r3, r2, r3
520: 607b      str     r3, [r7, #4]
522: 68bb      ldr     r3, [r7, #8]
524: 005b      lsls    r3, r3, #1
526: 687a      ldr     r2, [r7, #4]
528: 4413      add     r3, r2
52a: 60fb      str     r3, [r7, #12]
52c: 68f9      ldr     r1, [r7, #12]
52e: 4b05      ldr     r3, [pc, #20] ; (544 <main+0x38>)
530: 447b      add     r3, pc
532: 4618      mov     r0, r3
534: f7ff ef4a blx     3cc <printf@plt>
538: 2300      movs    r3, #0
53a: 4618      mov     r0, r3
53c: 3710      adds    r7, #16
53e: 46bd      mov     sp, r7
540: bd80      pop     {r7, pc}
542: bf00      nop
544: 00000064 .word   0x00000064

00000548 <__libc_csu_init>:
pa @clientssh-arm:~/ROTut/objdumpTut$
```

In diesem Fall kann also in der Ausgabe einfach nach main gesucht werden. Dort findet sich auch das als Beispiel gegebene Programm wieder. Sichtbar ist zum Beispiel, wie an Adresse 512 der Wert 4 in r3 geladen wird, dies entspricht `int a=4;`. Bei Adresse 524 ist die Addition mit 2 als logischer Shift nach links wiederzufinden. Sollten Mnemonics nicht klar sein, empfiehlt es sich, im [Reference Guide](#) unter Abschnitt C nachzuschlagen.

Nach diesem Schema können also die Instruktionen untersucht und analysiert werden.

Sollte es gewollt sein, können die (ELF) Fileheader folgendermaßen ausgegeben werden:

```
$> objdump -f sample
```

### 3.4 Hinweis: grep

Da die Menge an Output schnell frustrierend werden kann, wenn man eigentlich nur einen kleinen Abschnitt betrachten möchte, bietet es sich an, die Ausgabe in grep zu pipen. Die Ausgabe wird also an grep weitergeleitet, und dort kann beispielsweise nach main gesucht werden. Mit folgendem Befehl wird mittels `objdump sample` untersucht, allerdings wird das Ergebnis gefiltert, sodass nur die Stelle ab `<main>` und die nächsten 30 Zeilen sichtbar sind:

```
$> objdump -S sample | grep "<main>" -A 30
```