

Sichere Programmierung

Projekt 2

Einführung in Python

Nils Klein
(79373)

Alexander Krause
(79878)

Abdullah Yildiz
(79669)

Dozentin: Corina Hampel

Inhaltsverzeichnis

1	Teil 1: GDB Aufgabe 1	3
1.1	Teil A - Analyse des Codes	3
1.2	Teil B - Kompilieren des Programms	4
1.3	Teil C - Disassembly	5
1.4	Teil D - GDB	6
2	Teil 2: GDB Aufgabe 2	8
2.1	Teil A - Analyse des Codes	8
2.2	Teil B - Kompilieren des Programms	9
2.3	Teil C - Disassembly	10
3	Teil 3: GDB Aufgabe 3	12
3.1	Teil A - Analyse des Codes	12
3.2	Teil B - Kompilieren des Programms	12
3.3	Teil C - Disassembly	13
4	Teil 4: GDB Aufgabe 4	14
4.1	Teil A - Analyse des Codes	14
4.2	Teil B - Kompilieren des Programms	14
4.3	Teil C - Disassembly	15
4.4	Teil D - GDB	16
5	Teil 5: GDB Aufgabe 5	17
5.1	Teil A - Analyse des Codes	17
5.2	Teil B - Kompilieren des Programms	19
5.3	Teil C - Disassembly	20
5.4	Teil D - GDB	21

1 Teil 1: GDB Aufgabe 1

1.1 Teil A - Analyse des Codes

Das Skript "gdb-uebung-1.c" enthält nur eine main() Methode. Es wird die Bibliothek "stdio.h" importiert, und vorerst nur eine vorzeichenlose Ganzzahl Integer Variable deklariert:

```
1      #include <stdio.h>
2
3      int main(){
4          unsigned int i;
5          ..
6          ..
7      }
```

Die Bibliothek "stdio.h" bietet mehrere Funktionen für Dateioperationen an. Darunter fällt auch "printf" das im weiteren Verlauf des Codes verwendet wird.

Durch den Datentyp "unsigned" kann kein Überlauf der Zahlen von Positiv auf Negativ erfolgen. Bei einem signed short ist der maximale Wert "32768". Sobald nun um eins erhöht wird entsteht ein Überlauf, und es ist der Wert -32768 vorhanden. Durch **unsigned int** sind ausschließlich positive Zahlen möglich (0 - 4294967295).

Der kurze Code besitzt auch die folgende for-Schleife:

```
1      for(i=0; i<20; i++){
2          printf("i: %2d\n", i);
3      }
```

Die for-Schleife initialisiert als erstes "i" mit dem Wert "0". Die Bedingung der for-Schleife lautet, solange **i** kleiner als **20** ist, soll die Schleife ausgeführt werden. In der Schleife wird ein **printf** Befehl mit der Variable **i** ausgeführt. Die Funktion besteht aus einem Formatierungsteil und den konkret auszugebenden Argumenten. Durch die Hochklammern wird ein String ausgegeben, der zusätzlich folgendes Argument "%2d\n" enthält. Das Argument ist für die Ausgabe von zwei stelligen Integer-Werten verantwortlich. Nach der Formatierung wird durch ein "," der Wert der Variable "i" ausgegeben.

Nach diesen Erkenntnissen wird durch die for-Schleife die Variable **i** solange jeweils immer um 1 erhöht und ausgegeben, bis sie die Zahl 20 erreicht. Ist der Wert 20 erreicht wird das Programm beendet:

```
1      return 0;
```

1.2 Teil B - Kompilieren des Programms

In Aufgabe 1b sollte das Programm **gdb-uebung1.c** ausgeführt werden. Nach dem Kompilieren wird folgendes ausgegeben:

```
(gdb) run
Starting program: /home/tux/Praktikum_2/gdb-uebung-1
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
i: 10
i: 11
i: 12
i: 13
i: 14
i: 15
i: 16
i: 17
i: 18
i: 19
[Inferior 1 (process 2072) exited normally]
(gdb) █
```

Abbildung 1: Aufgabe 1b, Ausgabe des Terminals

Wie erwartet wird die Variable **"i"** bis zum Wert **19** ausgegeben, und bei **i = 20** das Programm beendet.

1.3 Teil C - Disassembly

In dieser Teilaufgabe soll nun erklärt werden, wie der Maschinencode genau aufgebaut ist: Der folgende Assemblercode kann in drei Abschnitte zerlegt werden:

```
(gdb) disass
Dump of assembler code for function main:
0x000055555555464a <+0>:    push    rbp
0x000055555555464b <+1>:    mov     rbp, rsp
=> 0x000055555555464e <+4>:    sub     rsp, 0x10
0x0000555555554652 <+8>:    mov     DWORD PTR [rbp-0x4], 0x0
0x0000555555554659 <+15>:   jmp     0x555555554675 <main+43>
0x000055555555465b <+17>:   mov     eax, DWORD PTR [rbp-0x4]
0x000055555555465e <+20>:   mov     esi, eax
0x0000555555554660 <+22>:   lea     rdi, [rip+0xad]          # 0x555555554714
0x0000555555554667 <+29>:   mov     eax, 0x0
0x000055555555466c <+34>:   call    0x555555554520 <printf@plt>
0x0000555555554671 <+39>:   add     DWORD PTR [rbp-0x4], 0x1
0x0000555555554675 <+43>:   cmp     DWORD PTR [rbp-0x4], 0x13
0x0000555555554679 <+47>:   jbe     0x55555555465b <main+17>
0x000055555555467b <+49>:   mov     eax, 0x0
0x0000555555554680 <+54>:   leave
0x0000555555554681 <+55>:   ret
End of assembler dump.
(gdb) █
```

Abbildung 2: Aufgabe 1c, Assemblercode

Im **ersten Abschnitt** ist die Initialisierung der Variable `i` und die `for`-Schleife enthalten. Die erste wichtige Zeile ist die Initialisierung der Variable `i` die in Zeile 8 ("`<+8>`") stattfindet. Dort wird die Variable im Stack mit dem **DWORD** Wert `0x0` angelegt. Der Datentyp "DWORD" ist ein 32-Bit unsigned Integer. Wird der Hexadezimalwert `0x0` in das Dezimalsystem umgerechnet, entspricht das dem Wert 0.

Die `for`-Schleife ist in Zeile 15 ("`<+15>`") zu finden. Dort ist der Befehl "**jmp**" zu sehen, dadurch springt der Instruction Pointer auf Zeile 43 und prüft mittels dem Befehl "**cmp**" (compare) ob die Variable `i` den gleichen Wert wie "**0x13**" hat. Der Dezimalwert der Hexadezimalen Zahl "`0x13`" entspricht 19. In der nächsten Zeile 47 wird auf die Zeile 17 "gesprungen", falls die Zahl kleiner oder gleich 19 ist. Dafür ist der Befehl "**jbe**" (Jump if Below or Equal) verantwortlich. Laut der Doku, sollte dieser Befehl verwendet werden, wenn man eine unsigned Integer Zahl vergleicht.

Im **zweiten Abschnitt** ist der Inhalt der `for`-Schleife enthalten. Jedoch sind die Zeilen 17 - 34 für die Funktionalität der Schleife nicht relevant.

Im **dritten Abschnitt** ist das Erhöhen der Variable `i` und der Vergleich der `for`-Schleife vorhanden. Für die Addition von `i` ist die Zeile 39 mit dem Befehl "**add**" verantwortlich. Hier wird der Wert `0x1` (also der Wert "1") zu dem Zieloperand "`i`" (`[rbp-0x4]`) addiert. Wie bereits erwähnt, wird der Vergleich in Zeile 43 durch "**cmp**" vorgenommen. Es wird der Inhalt von `DWORD PTR [rbp-0x4]` mit dem Wert `0x13` (19) verglichen. Ist die Zahl größer oder gleich `0x13`, wird die Schleife verlassen.

1.4 Teil D - GDB

Für diese Aufgabe wurde die Intel-Notation verwendet. Um nun das Programm zu übersetzen sind folgende Befehle nötig:

```
1 gcc -g -c gdb-uebung-1.c
2 gcc -o gdb-uebung-1 gdb-uebung-1.c
```

- Der erste Parameter **"-g"** führt zum Übersetzen des Programmes mit Debug Informationen.
- Der zweite Parameter **"-c"** kompiliert die Quelldatei.
- Der dritte Parameter **"-o"** gibt den Dateinamen der neuen Datei an, die erstellt wird. Dadurch kann das Programm mit dem Namen **"gdb-uebung-1"** aufgerufen werden.

Nun kann der GNU-Debugger verwendet werden:

```
1 gdb gdb-uebung-1
```

Die folgenden Befehle wurden in dieser Aufgabe verwendet:

- b main
- run main
- disass

```
1 b main = "break main", Breakpoint am Anfang der Funktion "main()".
2
3 run main = die Funktion main() wird ausgeführt.
4
5 disass = "disassemble", zerlegt eine Funktion oder ein Teil einer
6         Funktion.
7
8 nexti   = führt eine Instruktion aus und stoppt.
9         Gleiche Funktionalität wie "stepi".
10
11 print i = gibt den Wert der Variable "i" aus.
12
13 quit    = gdb verlassen.
```

Nach einem `b main`, `run main` und letztlich `disass` entsteht folgender Output:

```
(gdb) disass
Dump of assembler code for function main:
    0x000055555555464a <+0>:    push    rbp
    0x000055555555464b <+1>:    mov     rbp, rsp
=> 0x000055555555464e <+4>:    sub     rsp, 0x10
=> 0x0000555555554652 <+8>:    mov     DWORD PTR [rbp-0x4], 0x0
    0x0000555555554659 <+15>:   jmp     0x555555554675 <main+43>
=> 0x000055555555465b <+17>:   mov     eax, DWORD PTR [rbp-0x4]
    0x000055555555465e <+20>:   mov     esi, eax
    0x0000555555554660 <+22>:   lea     rdi, [rip+0xad]          # 0x555555554714
    0x0000555555554667 <+29>:   mov     eax, 0x0
    0x000055555555466c <+34>:   call    0x555555554520 <printf@plt>
    0x0000555555554671 <+39>:   add     DWORD PTR [rbp-0x4], 0x1
=> 0x0000555555554675 <+43>:   cmp     DWORD PTR [rbp-0x4], 0x13
=> 0x0000555555554679 <+47>:   jbe     0x55555555465b <main+17>
    0x000055555555467b <+49>:   mov     eax, 0x0
    0x0000555555554680 <+54>:   leave
    0x0000555555554681 <+55>:   ret
End of assembler dump.
(gdb) █
```

Abbildung 3: Aufgabe 1d, Assemblercode - "disass"

Der Pfeil links ist der **"Instruction Pointer"** und zeigt auf den Befehl der als nächstes ausgeführt wird. Auf diesem Bild ist der Instruction Pointer öfters zu sehen, weil das Bild bearbeitet wurde um nicht das gleiche Bild öfters zu verwenden. Natürlich ist der Instruction Pointer in einem `disass` nur einmal vorhanden.

In den ersten drei Zeilen werden die Adressen des Stack-Frames initialisiert (**rote Linie**).

Durch den Befehl **"nexti"** wird der nächste Befehl ausgeführt. In Zeile acht findet somit die Initialisierung der Variable **"i"** mit dem Wert **"0"** statt (**blaue Linie**).

Als nächstes wird die Variable **i** mit dem Hexadezimalwert **"0x13"** in Zeile 43 verglichen (**gelbe Linie**). Rechnet man die Hexadezimalzahl in eine Dezimalzahl um, ergibt sich die Zahl 19.

Falls die Zahl kleiner oder gleich 19 ist (**jbe**), wird in die Zeile 17 (**<+17>**) gesprungen **grüne Zeile**.

Nun wird erneut geprüft ob **i** kleiner gleich 19 ist (**weiße Linie**). Die Ausgaben werden bis **i < 20** ausgegeben (siehe Abbildung 1). Sobald **i** den Wert 20 erreicht, wird das Programm beendet.

2 Teil 2: GDB Aufgabe 2

2.1 Teil A - Analyse des Codes

Das Skript "gdb-uebung-2.c" enthält drei Funktionen und eine main() Methode. Es wird die Bibliothek "stdio.h" importiert:

```
1 #include <stdio.h>
2
3 int f(int a, int b) {
4     return 3*a + 7*b;
5 }
6
7 int g(int a, int b) {
8     return 10*a*a - 3*b;
9 }
10
11 int h(int a, int b) {
12     return a + b + 300;
13 }
14
15 int main() {
16     int a = 5, b=9, c=0;
17
18     c = f(g(a,h(a,b)),h(b,a));
19
20     printf("a = %d, b = %d, c = %d\n", a, b, c);
21 }
```

In Zeile 18 wird in die Variable "c", der Wert eines geschachtelten Funktionsaufrufes gespeichert. Im Anschluss wird eine Ausgabe von allen Variablen durchgeführt. Jedoch gibt es kein "return 0" in dem Quellcode.

2.2 Teil B - Kompilieren des Programms

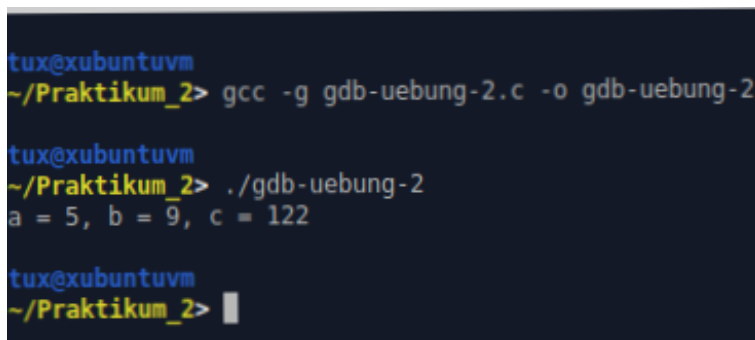
Die Ausgabe der Variablen a,b und c sollten folgende Werte haben:

- $a = 5$
- $b = 9$
- $c = 122$

Die zwei Variablen a und b sind nicht weiter zu erklären. Das Ergebnis von c leitet sich aus der Verschachtelung ab:

- Zu erst wird $h(a,b)$ aufgerufen. Ergebnis = 314
- Als nächstes wird $g(a, 317)$ aufgerufen. Ergebnis = -692
- Als nächstes wird $h(b, a)$ aufgerufen. Ergebnis = 314
- Als letztes wird nun $f(-692, 314)$ aufgerufen. Ergebnis = 122

Das Ergebnis ist wie erwartet:



```
tux@xubuntuv  
~/Praktikum_2> gcc -g gdb-uebung-2.c -o gdb-uebung-2  
  
tux@xubuntuv  
~/Praktikum_2> ./gdb-uebung-2  
a = 5, b = 9, c = 122  
  
tux@xubuntuv  
~/Praktikum_2> █
```

Abbildung 4: Aufgabe 2b, Ergebnis

2.3 Teil C - Disassembly

- In welcher Reihenfolge werden die Funktionen aufgerufen?

Das Programm ruft (Befehl: **call**) zuerst die Funktion $h(a,b)$ einmal auf. Durch die Verschachtelung wird die Funktion $h(b,a)$ erneut aufgerufen mit verdrehten Parametern. Danach kommt die Funktion $g(a,h(a,b))$ und als letztes die Funktion $f(g(a,h(a,b)),h(b,a))$.

```
Dump of assembler code for function main:
0x00005555555546b1 <+0>:    push    rbp
0x00005555555546b2 <+1>:    mov     rbp, rsp
0x00005555555546b5 <+4>:    push    rbx
0x00005555555546b6 <+5>:    sub     rsp, 0x18
=> 0x00005555555546ba <+9>:    mov     DWORD PTR [rbp-0x1c], 0x5
0x00005555555546c1 <+16>:   mov     DWORD PTR [rbp-0x18], 0x9
0x00005555555546c8 <+23>:   mov     DWORD PTR [rbp-0x14], 0x0
0x00005555555546cf <+30>:   mov     edx, DWORD PTR [rbp-0x1c]
0x00005555555546d2 <+33>:   mov     eax, DWORD PTR [rbp-0x18]
0x00005555555546d5 <+36>:   mov     esi, edx
0x00005555555546d7 <+38>:   mov     edi, eax
0x00005555555546d9 <+40>:   call    0x555555554698 <h>           1
0x00005555555546de <+45>:   mov     ebx, eax
0x00005555555546e0 <+47>:   mov     edx, DWORD PTR [rbp-0x18]
0x00005555555546e3 <+50>:   mov     eax, DWORD PTR [rbp-0x1c]
0x00005555555546e6 <+53>:   mov     esi, edx
0x00005555555546e8 <+55>:   mov     edi, eax
0x00005555555546ea <+57>:   call    0x555555554698 <h>           2
0x00005555555546ef <+62>:   mov     edx, eax
0x00005555555546f1 <+64>:   mov     eax, DWORD PTR [rbp-0x1c]
0x00005555555546f4 <+67>:   mov     esi, edx
0x00005555555546f6 <+69>:   mov     edi, eax
0x00005555555546f8 <+71>:   call    0x55555555466c <q>           3
0x00005555555546fd <+76>:   mov     esi, ebx
0x00005555555546ff <+78>:   mov     edi, eax
0x0000555555554701 <+80>:   call    0x55555555464a <f>           4
0x0000555555554706 <+85>:   mov     DWORD PTR [rbp-0x14], eax
0x0000555555554709 <+88>:   mov     ecx, DWORD PTR [rbp-0x14]
0x000055555555470c <+91>:   mov     edx, DWORD PTR [rbp-0x18]
0x000055555555470f <+94>:   mov     eax, DWORD PTR [rbp-0x1c]
0x0000555555554712 <+97>:   mov     esi, eax
0x0000555555554714 <+99>:   lea     rdi, [rip+0xa9]             # 0x5555555547c4
---Type <return> to continue, or q <return> to quit---
```

Abbildung 5: Aufgabe 2c, Reihenfolge der aufrufe

- Welche Stack Frames werden erzeugt?

Es wird für die `main()` und für die anderen Funktionen ein Frame erzeugt. Somit werden insgesamt zwei Frames erzeugt.

- Wie ist der Inhalt der Stack Frames?

Durch **"info frame"** können Informationen & Inhalt der Frames angezeigt werden:

```
(gdb) info frame
Stack level 0, frame at 0x7fffffffdf90:
 rip = 0x5555555546ba in main (gdb-uebung-2.c:16); saved rip = 0x7ffff7a05b97
 source language c.
 Arglist at 0x7fffffffdf80, args:
 Locals at 0x7fffffffdf80, Previous frame's sp is 0x7fffffffdf90
 Saved registers:
  rbx at 0x7fffffffdf78, rbp at 0x7fffffffdf80, rip at 0x7fffffffdf88
(gdb) █
```

Abbildung 6: Aufgabe 2c, Info Frame

3 Teil 3: GDB Aufgabe 3

3.1 Teil A - Analyse des Codes

Das Skript "gdb-uebung-3.c" enthält eine Funktion und eine main() Methode. Es wird die Bibliothek "stdio.h" importiert:

```
1 #include <stdio.h>
2
3 unsigned int f(unsigned int i) {
4     if (i>1) {
5         return i * f(i-1);
6     } else {
7         return 1;
8     }
9 }
10
11 int main() {
12     unsigned int i=5, r=0;
13
14     r = f(i);
15
16     printf("i = %d, f(i) = %d\n", i, r);
17 }
```

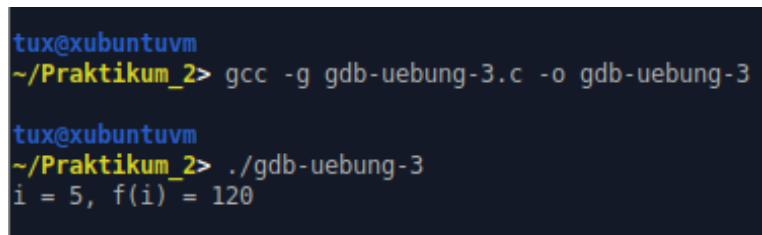
In der main Methode werden die zwei unsigned Integer Variablen i & r initialisiert. Im Anschluss wird durch eine Zuweisung zu r, die Funktion f mit dem Parameter i aufgerufen.

In der Funktion f wird zunächst geprüft ob i größer eins ist. Falls nicht, wird der das Programm mit einem "return 1" beendet. Andernfalls wird das Programm rekursiv aufgerufen, und i wird jedes mal um eins verringert.

Durch den printf in der main Methode wird der aktuelle Wert von i und das Ergebnis r ausgegeben. Es sollte für i=5 und für $r=5*4*3*2*1 = 120$ (Fakultät von 5) ausgegeben werden.

3.2 Teil B - Kompilieren des Programms

Das Ergebnis ist wie erwartet:



```
tux@xubuntvm
~/Praktikum_2> gcc -g gdb-uebung-3.c -o gdb-uebung-3

tux@xubuntvm
~/Praktikum_2> ./gdb-uebung-3
i = 5, f(i) = 120
```

Abbildung 7: Aufgabe 3b, Ergebnis

3.3 Teil C - Disassembly

- Wie viele Stack Frames werden erzeugt?

Es werden Insgesamt sechs Stack Frames erzeugt. Die main Methode hat einen Frame und für jeden Aufruf der Funktion f wird ein Frame erzeugt (siehe Abbildung).

```
(gdb) bt
#0  f (i=1) at gdb-uebung-3.c:9
#1  0x0000555555554668 in f (i=2) at gdb-uebung-3.c:5
#2  0x0000555555554668 in f (i=3) at gdb-uebung-3.c:5
#3  0x0000555555554668 in f (i=4) at gdb-uebung-3.c:5
#4  0x0000555555554668 in f (i=5) at gdb-uebung-3.c:5
#5  0x0000555555554695 in main () at gdb-uebung-3.c:14
```

- Wie ist der Inhalt dieser Stack Frames?

Der Inhalt besteht aus dem Parameter i der rekursiven Funktion (siehe Abbildung).

```
(gdb) frame 0
#0  f (i=1) at gdb-uebung-3.c:4
4      if (i>1) {
(gdb) frame 1
#1  0x0000555555554668 in f (i=2) at gdb-uebung-3.c:5
5      return i * f(i-1);
(gdb) frame 2
#2  0x0000555555554668 in f (i=3) at gdb-uebung-3.c:5
5      return i * f(i-1);
(gdb) frame 3
#3  0x0000555555554668 in f (i=4) at gdb-uebung-3.c:5
5      return i * f(i-1);
(gdb) frame 4
#4  0x0000555555554668 in f (i=5) at gdb-uebung-3.c:5
5      return i * f(i-1);
(gdb) frame 5
#5  0x0000555555554695 in main () at gdb-uebung-3.c:14
14     r = f(i);
```

- Wie wird die Parameterübergabe in Assembler umgesetzt?

Die System V AMD64 ABI wird unter Linux, FreeBSD und macOS befolgt. Somit ist dies ein Standard unter Unix und Unix-ähnlichen Betriebssystemen. Die ersten sechs Ganzzahl oder Zeigerargumente werden in den Registern: RDI, RSI, RDX, RCX, R8, R9 übergeben. Für Gleitkommazahlen werden folgende Register verwendet: XMM0 bis XMM7.

4 Teil 4: GDB Aufgabe 4

4.1 Teil A - Analyse des Codes

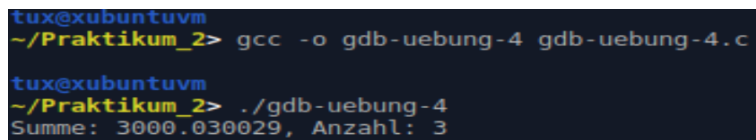
Das Skript "gdb-uebung-4.c" enthält eine nur eine main() Methode.

Durch eine for-Schleife sollen die Variablen **summe** und **anzahl** berechnet werden. Die Variable i hat einen Wert von 1000. Es soll bei jedem Schleifendurchlauf der Wert 0.01 zur Variable summe addiert werden. Wenn i den Wert 1000.04 erreicht, soll das Programm mit einem "return 0" beendet werden. Demnach sollten es vier Schleifendurchläufe geben.

```
1 #include <stdio.h>
2
3 int main() {
4     int anzahl;
5     float summe;
6     float i;
7
8     summe = 0, anzahl = 0;
9     for (i = 1000; i <= 1000.03; i += .01) {
10         summe += i;
11         anzahl++;
12     }
13     printf("Summe: %f, Anzahl: %d\n", summe, anzahl);
14
15     return 0;
16 }
```

4.2 Teil B - Kompilieren des Programms

Das Ergebnis ist nicht wie erwartet:



```
tux@xubuntuvn
~/Praktikum_2> gcc -o gdb-uebung-4 gdb-uebung-4.c
tux@xubuntuvn
~/Praktikum_2> ./gdb-uebung-4
Summe: 3000.030029, Anzahl: 3
```

Abbildung 8: Aufgabe 4b, Ergebnis

4.3 Teil C - Disassembly

- Berechnet die Schleife das korrekte Ergebnis?

Die Schleife berechnet nicht das korrekte Ergebnis. Da mit dem Datentyp "float" (Fließkommazahlen) gearbeitet wird, ist davon auszugehen dass Rundungsfehlern entstehen können. Das richtige Ergebnis sollte 4000.06 sein.

- Welche Werte nehmen die Variablen bei der Ausführung des Programms an?

Am Anfang ist $\text{anzahl}=0$, $\text{summe}=0$ und $i=1000$. In den folgenden Abbildungen sind die Werte der Schleifendurchgänge zu sehen.

Erster Schleifendurchlauf:

```
(gdb) info locals
anzahl = 1
summe = 1000
i = 1000.01001
```

Abbildung 9: Aufgabe 4c, Teil-Ergebnis

Zweiter Schleifendurchlauf:

```
(gdb) info locals
anzahl = 2
summe = 2000.01001
i = 1000.02002
```

Abbildung 10: Aufgabe 4c, Teil-Ergebnis

Dritter Schleifendurchlauf:

```
anzahl = 3
summe = 3000.03003
i = 1000.03003
```

Abbildung 11: Aufgabe 4c, Ergebnis

- Warum wird das falsche Ergebnis berechnet?

Durch die Bedingung $i \leq 1000.03$ müsste die Schleife ein viertes mal ausgeführt werden, sobald i den Wert 1000.03 erreicht. Durch den Rundungsfehler ist jedoch $i > 1000.03$, und die Schleife wird kein viertes mal ausgeführt. Dadurch wird die letzte Addition nicht vorgenommen, und der richtige Wert kommt somit nicht zustande.

4.4 Teil D - GDB

Sobald der Datentyp "double" anstatt "float" verwendet wird, wird die Schleife genau vier mal ausgeführt und liefert das richtige Ergebnis. Der Grund hierfür ist die 64 Bit Darstellung der Zahlen von double anstatt der 32 Bit Darstellung von float.

```
1
2 #include <stdio.h>
3
4 int main() {
5     int anzahl;
6     double summe;
7     double i;
8
9     summe = 0, anzahl = 0;
10    for (i = 1000; i <= 1000.03; i += .01) {
11        summe += i;
12        anzahl++;
13    }
14    printf("Summe: %f, Anzahl: %d\n", summe, anzahl);
15
16    return 0;
17 }
```

Der Vergleich von Gleitkommazahlen kann wie in der Aufgabe dargestellt nicht immer funktionieren. Das liegt konkret an der Berechnung von float Werten. Jede Variable des Typs float besteht aus einem Vorzeichenbit, einigen Bits die den Exponenten repräsentieren und Bits die die Mantisse darstellen. Dadurch kann float keine beliebigen realen Zahlen speichern, da diese mit einer Formel berechnet werden müssen. Zum Beispiel würde der Wert 0.1 als 0.100000001490116119384765625 dargestellt werden, das dem 0,1 nächstliegenden 32-Bit Gleitkommawert entspricht. Deshalb ist vom Datentyp float abzuraten sobald es um Nachkommastellen vergleiche geht!

5 Teil 5: GDB Aufgabe 5

5.1 Teil A - Analyse des Codes

Das Skript **"gdb-uebung-5.c"** besitzt eine `binarysearch()` Methode die sich rekursiv aufruft und eine `main()` Methode. Im Prinzip sucht die `binarysearch()` nach einem Element (Übergabeparameter `"Zahl"`). Bei jedem Durchlauf der Methode wird die Rekursionstiefe um eins erhöht und in die Variable `"rekursionstiefe"` gespeichert. Innerhalb der Methode gibt es vier verschiedene Möglichkeiten:

1. Wenn die übergebene `"Zahl"` gleich dem Wert der Stelle `"mitte"` ist, wird der Wert von `"mitte"` zurückgegeben.

2. Falls die Werte `"links"` und `"rechts"` im Binärbaum dem gleichen Wert entsprechen, wird der Wert `"-1"` zurückgegeben.

Dadurch wird folgendes ausgegeben: `"Zahl $zahl nicht gefunden!"`

3. Falls der Wert von `"mitte"` echt größer ist als die übergebene `"Zahl"`, wird die Methode Rekursiv mit den gleichen Werten erneut aufgerufen.

4. Falls der Wert von `"mitte"` nicht echt größer ist als die übergebene `"Zahl"`, wird ebenfalls die Methode Rekursiv aufgerufen, jedoch wird hier für den `"links"` Parameter der Wert von `"mitte"` genommen.

Der `main()` Methode wird ein Integer Wert `argc`, und ein Zeiger auf die Kommandozeilenparameter (`argv`) in einem Array mitgegeben. In der nächsten Zeile wird geprüft, ob weniger als zwei Parameter übergeben worden sind. Falls dies zutrifft, gibt das Programm eine Ausgabe aus, und beendet das Programm mit `"return 1"`.

In Zeile 30 wird der zweite Übergabeparameter (`argv[1]`) von einem String in eine Integer Zahl konvertiert durch den Befehl `atoi` (aus der **"stdlib.h"** Bibliothek), und in die Integer Variable `"Zahl"` gespeichert. Im Anschluss kommt eine for-Schleife, die solange `i` kleiner als die Anzahl des Arrays ist, eine Ausgabe macht.

Als letztes wird durch eine `If else` Abfrage eine Ausgabe ausgegeben. Falls die Position größer gleich 0 ist, wird die gesuchte Zahl und die Position im Array ausgegeben. Andernfalls wird ausgegeben, dass die Zahl nicht gefunden wurde. Durch ein **"return 0"** wird das Programm beendet.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define MAX 8
5 int array[MAX] = {1,4,12,18,26,31,40,42};
6 int rekursionstiefe = 0;
7
8 int binarysearch(int zahl, int links, int rechts) {
9     rekursionstiefe++;
10    int mitte = (links + rechts) / 2;
11    printf("\nRekursionstiefe: %d", rekursionstiefe);
12    if (array[mitte] == zahl)
13        return mitte;
14    if (links == rechts)
15        return -1;
16    if (array[mitte] > zahl)
17        return binarysearch(zahl, links, mitte);
18    else
19        return binarysearch(zahl, mitte, rechts);
20 }
21
22 int main(int argc, char *argv[]) {
23     int zahl, position, i;
24     if(argc < 2) {
25         printf("Benutzung: %s <zu suchende Zahl>\n", argv[0]);
26         return 1;
27     }
28     zahl = atoi(argv[1]);
29
30     for (i = 0; i < MAX; i++) {
31         printf("%4d", array[i]);
32     }
33     position = binarysearch(zahl, 0, 7);
34     if (position >= 0) {
35         printf("\nGesuchte Zahl %d an Arrayposition %d\n",
36                                     zahl, position);
37     }
38     else {
39         printf("\nZahl %d nicht gefunden\n", zahl);
40     }
41     return 0;
42 }

```

5.2 Teil B - Kompilieren des Programms

Bei den Werten 1, 4 und 26 liefert das Programm die entsprechenden Werte. Bei den Werten 27 und 42 endet das Programm in einer Endlosschleife und liefert eine Fehlermeldung:

```
tux@xubuntuvm
~/Praktikum_2> ./gdb-uebung-5 1
 1   4  12  18  26  31  40  42
Rekursionstiefe: 1
Rekursionstiefe: 2
Rekursionstiefe: 3
Gesuchte Zahl 1 an Arrayposition 0
```

Abbildung 12: Aufruf mit dem Parameter 1

```
tux@xubuntuvm
~/Praktikum_2> ./gdb-uebung-5 4
 1   4  12  18  26  31  40  42
Rekursionstiefe: 1
Rekursionstiefe: 2
Gesuchte Zahl 4 an Arrayposition 1
```

Abbildung 13: Aufruf mit dem Parameter 4

```
tux@xubuntuvm
~/Praktikum_2> ./gdb-uebung-5 26
 1   4  12  18  26  31  40  42
Rekursionstiefe: 1
Rekursionstiefe: 2
Rekursionstiefe: 3
Gesuchte Zahl 26 an Arrayposition 4
```

Abbildung 14: Aufruf mit dem Parameter 26

```
Rekursionstiefe: 174510
Rekursionstiefe: 174511
Rekursionstiefe: 174512
Rekursionstiefe: 174513
Rekursionstiefe: 174514
Rekursionstiefe: 174515
Rekursionstiefe: 174516
Rekursionstiefe: 174517
Speicherzugriffsfehler (Speicherabzug geschrieben)
```

Abbildung 15: Aufruf mit dem Parameter 27

```
Rekursionstiefe: 174553
Rekursionstiefe: 174554
Rekursionstiefe: 174555
Rekursionstiefe: 174556
Rekursionstiefe: 174557
Rekursionstiefe: 174558
Rekursionstiefe: 174559
Rekursionstiefe: 174560
Speicherzugriffsfehler (Speicherabzug geschrieben)
```

Abbildung 16: Aufruf mit dem Parameter 42

5.3 Teil C - Disassembly

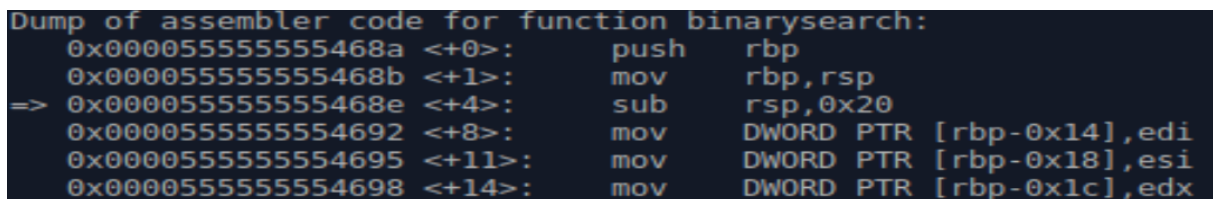
- An welcher Stelle liegt eine fehlerhafte Programmierung vor?

In der letzten If-Schleife.

Die Werte für links und rechts werden im **else** Block zurückgesetzt, sobald die Variable "mitte = 0" erreicht und der Knoten kleiner ist als die Zahl. Andersherum, wenn die Zahl gleich dem Maximum Knoten entspricht vergleicht Sie die Zahlen nicht und es entsteht ebenfalls eine Endlosschleife.

- Aus welchem Grund bricht das Programm ab?

Für jeden Funktionsaufruf wird ein Stack Frame angelegt (siehe Abbildung):



```
Dump of assembler code for function binarysearch:
0x000055555555468a <+0>:      push    rbp
0x000055555555468b <+1>:      mov     rbp, rsp
=> 0x000055555555468e <+4>:      sub     rsp, 0x20
0x0000555555554692 <+8>:      mov     DWORD PTR [rbp-0x14], edi
0x0000555555554695 <+11>:     mov     DWORD PTR [rbp-0x18], esi
0x0000555555554698 <+14>:     mov     DWORD PTR [rbp-0x1c], edx
```

Abbildung 17: Aufgabe 5c, Stack Frames

Falls die Funktion nun in einer Endlosschleife hängt, werden unendlich viele Stack Frames erstellt. Folglich läuft der reservierte Speicher für das Programm voll. Sobald das Programm kein Speicher mehr zu Verfügung hat, bricht das Programm mit der Fehlermeldung "Speicherzugriffsfehler ab".

5.4 Teil D - GDB

Das Original Programm hat folgende Fälle nicht geprüft:

- Ob die gesuchte Zahl zwischen zwei Knoten liegen würde, aber nicht existiert.
- Ob die gesuchte Zahl gleich dem Wert des maximal Knotens entspricht.
- Ob die gesuchte Zahl größer ist als der Wert des maximal Knotens.

Um das Programm simpel wie möglich zu halten, wurden die bestehenden Abfragen um weitere If-else Abfragen erweitert. Die entsprechenden If-else Abfragen wurden für den unteren und oberen Bereich des Arrays implementiert, und natürlich mit verschiedenen Werten getestet.

```
1 int binarysearch(int zahl, int links, int rechts) {
2     rekursionstiefe++;
3     int mitte = (links + rechts) / 2;
4     printf("\nRekursionstiefe: %d", rekursionstiefe);
5     if (array[mitte] == zahl)
6         return mitte;
7     if (links == rechts)
8         return -1;
9     if (array[mitte] > zahl) //untere Knoten
10        //Wert liegt zwischen 2 Knoten, ist aber nicht im Array
11        if(array[mitte-1] < zahl)
12            return -1;
13        else
14            return binarysearch(zahl, links, mitte);
15    else //obere Knoten
16        //Zahl ist gleich Max-Knoten
17        if(array[rechts] == zahl)
18            return rechts;
19        //Zahl ist größer Max-Knoten
20        else if (array[rechts] < zahl)
21            return -1;
22        else
23            //Wert liegt zwischen 2 Knoten, ist aber nicht im Array
24            if(array[rechts-1] < zahl)
25                return -1;
26            else
27                return binarysearch(zahl, mitte, rechts);
28 }
```