

## Inhalt:

1. crash.c - Speicherfehler .....	1
2. class.cpp - Variablen .....	2
3. power.c - Stepping und Laufzeitinformationen .....	4
4. endless.c - Prozesse und Variablenbeobachtung .....	7

## Hinweise:

- Die Ausgaben sind teilweise formatiert, um die Lesbarkeit zu fördern. Zudem wurden unwichtige Zeilen entfernt.
- Da es sich um Laufzeitinformationen handelt, können die Adressen beim Nachvollziehen der Beispiele anders ausfallen.
- Bei sehr ähnlichen Zeilen wurde oftmals die Erklärung bzw. Ausgabe weggelassen.

---

## 1. crash.c - Speicherfehler

---

### 1.1. Source code:

```
1: #include <stdio.h>
2:
3: int main()
4: {
5:     int* ptr = NULL;
6:
7:     *ptr = 123; // Ungueltige Referenzierung
8:     printf("Nach Zuweisung\n");
9:     return 0;
10: }
```

## 1.2. Demonstration von:

Das erste Beispiel zeigt, wie einfach es ist, mittels des GDB, Speicherfehlern auf die Schliche zu kommen.

## 1.3. Befehlsfolge:

---

1.	Befehl:	<code>./crash</code> (Kommandozeile)
	Erklärung:	Ein typisches Problem: Das Programm stürzt ab und man weiß nicht warum.
2.	Befehl:	<code>gdb ./crash</code>
	Erklärung:	GDB wird mit “./crash” als Executable aufgerufen.
3.	Befehl:	<code>list 1</code>
	Ausgabe:	<komplettes Source listing>
	Erklärung:	Ausgabe der Zeilen 1-11.
4.	Befehl:	<code>run</code>
	Ausgabe:	Program received signal EXC_BAD_ACCESS, Could not access memory. 0x00001d78 in main() at 01-crash.c:7 7 *ptr = 123; // Ungueltige Referenzierung
	Erklärung:	Das Programm wird gestartet und hält seine Ausführung an der Zeile 7 wegen eines Speicherzugriffsfehlers an. Crash-Ursache gefunden: NULL-Pointer Schreibzugriff.

---

## 2. class.cpp - Variablen

---

### 2.1. Source code:

```
1: #include <iostream>
2: using namespace std;
3:
4: static const int MAX_NUM = 2;
5:
6: class TestClass {
7:     struct TestStruct {
8:         int x, y;
9:     } structVar[MAX_NUM];
10:
11:     friend ostream& operator<<(ostream& os, const TestClass& tc);
12: public:
13:     void setVar(int nr, int x, int y);
14: };
15:
```

```

16: void TestClass::setVar(int nr, int x, int y)
17: {
18:     structVar[nr].x = x;
19:     structVar[nr].y = y;
20: }
21:
22: ostream& operator<<(ostream& os, const TestClass& tc)
23: {
24:     for (int i = 0; i < MAX_NUM; i++)
25:         os << i << ". x=" << tc.structVar[i].x << ", y=" <<
tc.structVar[i].y << endl;
26:     return os;
27: }
28:
29: int main()
30: {
31:     TestClass* tc = new TestClass();
32:     tc->setVar(0, 1, 2);
33:     tc->setVar(1, 2, 3);
34:
35:     cout << "Ausgabe: " << *tc;
36:
37:     delete tc;
38:     return 0;
39: }

```

## 2.2. Demonstration von:

Auch komplexe, verschachtelte Datenstrukturen machen dem GDB keine Schwierigkeiten. So können mittels “print” die Werte aller sichtbaren Variablen ausgegeben werden.

Außerdem eine erste Demonstration, wie man mittels Breakpoints ein Programm an einer bestimmten Stelle - hier Zeile - anhält.

## 2.3. Befehlsfolge:

1.	Befehl: gdb
	Erklärung: Starten des GDB ohne die zu debuggende Datei anzugeben.
2.	Befehl file ./class
	Erklärung: Nachträgliches Einlesen der Executable und dazugehörigen Symbole.
3.	Befehl: whatis MAX_NUM
	Ausgabe: type = const int
	Erklärung: Mittels “whatis” kann man sich den Typ einer Variable bzw. Konstante ausgeben lassen.
4.	Befehl: break 35
	Ausgabe: Breakpoint 1 at 0x2bb4: file 02-class.cpp, line 35.
	Erklärung: Setzt einen Breakpoint auf Zeile 35, diese entspricht der Adresse 0x2bb4.

5.	Befehl:	info breakpoints
	Ausgabe:	Num Type           Disp Enb Address           What 1 breakpoint keep y 0x00002bb4 in main at 02- class.cpp:35
	Erklärung:	Anzeige der gesetzten Breakpoints.
6.	Befehl:	run
	Ausgabe:	Starting program: ../../class [Switching to process 1091 thread 0xb03] : Breakpoint 1, main () at class.cpp:35 35 cout << "Ausgabe: " << *tc;
	Erklärung:	Startet die Programmausführung (Prozess 1091) und stoppt bei Zeile 35 die Ausführung.
7.	Befehl:	print MAX_NUM
	Ausgabe:	\$1 = 2
	Erklärung:	Ausgabe des akt. Wertes ("2") von MAX_NUM
8.	Befehl:	info locals
	Ausgabe:	tc = (TestClass *) 0x68400
	Erklärung:	Alle sichtbaren lokalen Variablen - hier lediglich "tc"
9.	Befehl:	print tc
	Ausgabe:	\$1 = (TestClass *) 0x68400
	Erklärung:	Ausgabe des Inhalts der Variable "tc".
10.	Befehl:	print *tc
	Ausgabe:	\$2 = { structVar = { { x = 1, y = 2 }, { x = 2, y = 3 } } }
	Erklärung:	Kompletter Inhalt der Instanz "tc" durch Dereferenzieren.
11.	Befehl:	print tc->structVar[1].x
	Ausgabe:	\$3 = 2
	Erklärung:	Ausgabe - Zugriff wie in C.

### 3. power.c - Stepping und Laufzeitinformationen

#### 3.1. Source code:

```

1: #include <stdio.h>
2: #include <stdlib.h>
3:
4: int power(int basis, int ex)
5: {
6:     if (ex == 0) return 1;
7:     return basis*power(basis, ex-1);
8: }

```

```

9:
10: int main(int argc, char** argv)
11: {
12:     int basis = 0, ex = 0;
13:
14:     if (argc != 3) {
15:         printf("%s basis exponent\n", argv[0]);
16:         return 1;
17:     }
18:     basis = (int)strtol(argv[1], NULL, 10);
19:     ex = (int)strtol(argv[2], NULL, 10);
20:     printf("--> Ergebnis von %d^%d = %d\n", basis, ex, power(basis,
ex));
21:     return 0;
22: }

```

### 3.2. Demonstration von:

Durch zeilenweises Ausführen findet man Fehler (z.B. Rechenfehler) oftmals einfacher, u.a. weil man sich in Kombination mit dem oben bereits benutzten "print" die akt. Werte ausgeben lassen kann. Auch sieht man, welche Sprünge gemacht wurden.

Um herauszufinden, wie man in die akt. Funktion gekommen ist, sind sog. "Backtraces" nützlich.

### 3.3. Befehlsfolge:

1.	Befehl:	gdb ./power
2.	Befehl:	break 18
	Erklärung:	Programm soll vor der "basis" Zuweisung anhalten.
3.	Befehl:	run
	Ausgabe:	: Program exited with code 01.
	Erklärung:	Das Programm wurde beendet ohne jemals den Breakpoint Zeile 18 zu erreichen. Die Erklärung liefert der Exitcode 01 - Zeile 16 wurde das exit() aufgerufen.
4.	Befehl:	info breakpoints
	Ausgabe:	Num Type Disp Enb Address What 1 breakpoint keep y 0x00001cdc in main at power.c:18
	Erklärung:	Breakpoints bleiben auch nach dem Durchlauf eines Programmes erhalten.
5.	Befehl:	set args 3 5
	Erklärung:	Kommandozeilenparameter (basis = "3", exponent = "5") setzen.
6.	Befehl:	run
	Ausgabe:	Breakpoint 1, main (argc=3, argv=0xbffffd44) at 03- power.c:18 18 basis = (int)strtol(argv[1], NULL, 10);

7.	Befehl:	<code>info args</code>	
	Ausgabe:	<code>argc = 3</code> <code>argv = (char **) 0xbffffd44</code>	
	Erklärung:	Der akt. Funktion (“main”) wurde mit den Parametern “argc” und “argv” aufgerufen.	
8.	Befehl:	<code>print basis</code>	
	Ausgabe:	<code>\$1 = 0</code>	
	Erklärung:	Die Variable “basis” mit dem Initialwert 0 wird ausgegeben.	
9.	Befehl:	<code>next</code>	
	Ausgabe:	<code>19 ex = (int)strtol(argv[2], NULL, 10);</code>	
	Erklärung:	Auswerten der 18. Zeile und danach sofortiges Stoppen des Programmflusses.	
10.	Befehl:	<code>print basis</code>	
	Ausgabe:	<code>\$2 = 3</code>	
	Erklärung:	Nach dem Auswerten der 18. Zeile ist “3” der Inhalt von “basis”.	
11.	Befehl:	<code>info breakpoints</code>	
	Ausgabe:	<code>1 breakpoint keep y 0x00001cdc in main at 03-power.c:18</code> <code>breakpoint already hit 1 time</code>	
	Erklärung:	Der GDB zählt wieviel mal ein Breakpoint bereits ausgeführt wurde.	
12.	Befehl:	<code>delete 1</code>	
	Erklärung:	Entfernt den Breakpoint mit Nr. 1 (= Zeile 18)	
13.	Befehl:	<code>break 6 if (ex == 2)</code>	
	Erklärung:	Sobald der Parameter “ex = 2” ist, soll bei Zeile 6 angehalten werden.	
14.	Befehl:	<code>continue</code>	
	Ausgabe:	<code>Breakpoint 2, power (basis=3, ex=2) at 03-power.c:6</code> <code>6 if (ex == 0) return 1;</code>	
	Erklärung:	Die Programmausführung wird bis zum aktiv Werden des 6. Breakpoints fortgesetzt. Nützlich ist die automatische Anzeige der Funktionsargumente.	
15.	Befehl:	<code>info frame</code>	
	Ausgabe/ Erklärung:	Anzeige des Stacks und div. anderer zugehörigen Informationen.	
16.	Befehl:	<code>backtrace</code>	
	Ausgabe:	<code>#0 power (basis=3, ex=2) at 03-power.c:6</code> <code>#1 0x00001c54 in power (basis=3, ex=3) at 03-power.c:7</code> <code>#2 0x00001c54 in power (basis=3, ex=4) at 03-power.c:7</code> <code>#3 0x00001c54 in power (basis=3, ex=5) at 03-power.c:7</code> <code>#4 0x00001d20 in main (argc=3, argv=0xbffffd44) at 03-power.c:20</code>	
	Erklärung:	Aufrufreihenfolge - der letzte oben - der Funktionen inkl. den jeweiligen Parametern.	
17.	Befehl:	<code>set basis = 2</code>	
	Erklärung:	Der Variable “basis” den Wert “2” zuweisen.	
18.	Befehl:	<code>print basis</code>	
	Ausgabe:	<code>\$3 = 2</code>	

19.	Befehl:	<code>set basis = 3</code>
	Erklärung:	Zurücksetzen von "basis" auf den ursprünglichen - unveränderten - Wert.
20.	Befehl:	<code>4x finish</code>
	Ausgabe:	Hier nur die Ausgabe nach dem ersten "finish": : 7 return basis*power(basis, ex-1); Value returned is \$4 = 9
	Erklärung:	Die akt. Funktion zu Ende laufen lassen und dann wieder stoppen und den Rückgabewert ausgeben.

---

## 4. endless.c - Prozesse und Variablenbeobachtung

---

### 4.1. Source code:

```

1: #include <stdio.h>
2:
3: static var;
4:
5: int endless()
6: {
7:     int loop = 0, var = 0;
8:
9:     for (;;) {
10:         if (loop == 100) {
11:             printf("\r%c", ((var % 2) ? '-' : '+'));
12:             var++;
13:             loop = 0;
14:         }
15:         loop++;
16:     }
17:     return var; // Niemals erreicht
18: }
19:
20: int main(int argc, char** argv)
21: {
22:     int ret = endless();
23:     printf("rueckgabewert: %d\n", ret);
24:     return 0;
25: }

```

## 4.2. Demonstration von:

Eine Applikation läuft eine Zeit lang stabil, doch irgendwann produziert sie Fehler. Ein Neustarten kommt nicht in Frage, weil sonst der “fehlerhafte Zustand” wieder verlassen wird. Deshalb kann man auf bereits laufende Prozesse debuggen.

Sog. “Watches” lassen es zu, das man über jeglichen Zugriff auf eine Variable informiert wird.

## 4.3. Befehlsfolge:

1.	Befehl:	<code>./endless</code> (in einem anderen Terminalfenster)
	Ausgabe:	<Abwechselnd “+” und “-“>
2.	Befehl:	<code>gdb ./endless</code>
3.	Befehl:	<code>info scope 22</code>
	Ausgabe:	Symbol argc is an argument at stack/frame offset 120 length 4. Symbol argv is an argument at stack/frame offset 124, length 4. Symbol ret is a local variable at frame offset 64,length 4
	Erklärung:	Ausgabe der sichtbaren Variablen.
4.	Befehl:	<code>shell ps grep endless</code>
	Ausgabe:	1163 pl R+ 0:09.61 ./04-endless 1168 std S+ 0:00.01 bash -c ps grep endless
	Erklärung:	Führt den Shell Befehl “ps grep endless” aus.
5.	Befehl:	<code>define ps</code> <code>shell ps grep \$arg0</code> <code>end</code>
	Erklärung:	Definiert das User-Kommando “ps”.
6.	Befehl:	<code>ps endless</code>
	Ausgabe:	<Wie bei 4.>
	Erklärung:	Aufruf des User-Kommandos “ps” mit \$arg0 = “endless”.
7.	Befehl:	<code>attach 1163</code>
	Ausgabe:	Attaching to program: `././ endless', process 1163. : 0x00001ce0 in endless () at 04-endless.c:10 10 if (loop == 100) {
	Erklärung:	Der GDB hängt sich in den Programmfluß von Prozess 1163 und stoppt ihn sofort - die Zeile des Stoppes ist beliebig, kann auch innerhalb einer Systemlibrary sein. Man sieht auch, das im anderen Terminalfenster kein Zeichenwechsel mehr stattfindet.
8.	Befehl:	<code>display loop</code>
	Ausgabe:	1: loop = 44
	Erklärung:	Es wird ein Display auf “loop” gesetzt. Der akt. Wert von “loop” ist “44”.
9.	Befehl:	<code>display var</code>
	Ausgabe:	2: var = 7919288

10.	<b>Befehl:</b>	<code>watch var</code>
	<b>Erklärung:</b>	Stoppen, sobald "var" verändert wird - sprich schreibend zugegriffen.
11.	<b>Befehl:</b>	<code>continue</code>
	<b>Ausgabe:</b>	<pre>1: loop = 100 2: var = 7919289  Old value = 7919288 New value = 7919289</pre>
	<b>Erklärung:</b>	Bei Zugriff auf die Schreibzugriff auf "var" - Zeile 12 - wird das Programm angehalten. Es werden außerdem die Displays ausgeführt, d.h. der akt. Inhalt angezeigt.
12.	<b>Befehl:</b>	<code>return 23</code>
	<b>Ausgabe:</b>	<pre>Make endless return now? (y or n) y #0 0x00001d68 in main (argc=1, argv=0xbffffd34) at 04- endless.c:22 22 int ret = endless();</pre>
	<b>Erklärung:</b>	Sofortiger Abbruch der Funktion "endless" mit dem Rückgabewert "23".