Seminar
Effiziente Programmierung
Debugging und Speicherfehler

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Name: Kadir Duman
E-Mail-Adresse: 4duman@informatik.uni-hamburg.de
Matrikelnummer: 6662793
Studiengang: Software System Entwicklung
Betreuer: Michael Kuhn
Hamburg, den 03.03.2017

Inhalt

1 Einleitung

In diesem Seminar geht es um „effiziente Programmierung“. Es ist so, dass in Programmen, in der Regel, immer Fehler auftauchen. Es kann dadurch passieren, wenn zum Beispiel neue Funktionen implementiert und hinzugefügt werden, das Quellcode durch einen kompakteren Quellcode ersetzt wird, oder eine Library verändert wird. Dafür ist es wichtig, dass Programmierer Methoden kennen, wie Fehler im Programm erkannt und behoben werden können. Zudem wäre es vorteilhaft, wenn das Erkennen und Beheben von Fehlern schnell passieren kann.

Im Folgenden wird zuerst das Thema „Debugging“ beleuchtet, damit der Leser das Grundlegende, wie Fehler gefunden werden können, versteht.

Anschließend folgt das Thema „Speicherfehler in C“. In diesem Thema geht es um die typischen Speicherfehler, die beim Programmieren in der Sprache „C“ entstehen können. Durch diesen Abschnitt soll dem Leser deutlich gemacht werden, worauf geachtet werden muss, damit solche Fehler nicht passieren.
Trotzdem kann es vorkommen, dass Speicherfehler auftreten. Um diese Fehler zu finden, gibt es die so genannten „Sanitizer“.

Wie weiter oben auch erwähnt, stellt sich nun die Frage, ob und wie Fehler effizient gefunden werden können. Für die Beantwortung dieser Frage, werden die Themen „Delta Debugging“ und „Continuous Testing“ behandelt. Anschließend wird das Werkzeug Valgrind, das überhaupt nicht effizient ist, näher erläutert.

2 Debugging

2.1 Was ist ein Bug?

Ein Bug ist ein Programmfehler, welcher zur Laufzeit (erst bei der Ausführung) auftritt. Solche Fehler nennt man daher Laufzeitfehler. Nachdem ein Bug aufgetreten ist, zeigt das Programm ein fehlerhaftes Verhalten. Die Korrektheit des Programms ist gefährdet, indem zum Beispiel, falsche Werte übergeben werden.
Möglicherweise wird die Ausführung des Programms komplett angehalten [2].
Eine Fehlermeldung könnte Hinweise auf die Ursache geben, jedoch kommt es oft vor, dass die Ursache nicht offensichtlich ist.
Sobald die Ursache nicht offensichtlich ist und somit der Fehler nicht direkt gefunden und behoben werden kann, müssen Hilfsmittel wie Debugging benutzt werden.

2.2 Was ist Debugging?

Sobald deutlich wird, dass ein Programm fehlerhaft ist, muss der Fehler zuerst lokalisiert werden. Es wird versucht den Teil des Programms zu finden, der für den Fehler verantwortlich ist.
Nach der Fehlerlokalisierung muss der Programmierer nun den Fehler beheben, indem er den fehlerhaften Teil des Programms verändert. Dieser Vorgang wird auch „Bugfixing“ genannt.

Debugging besteht also aus den Aufgaben Fehlerlokalisierung und Bugfixing.
Die Fehlerlokalisierung kann viel Zeit in Anspruch nehmen, denn wenn ein Programm hunderttausende Zeilen an Code hat, kann es etwas länger dauern bis der fehlerhafte Teil gefunden wird.
Die zweite Aufgabe Bugfixing kann dagegen kompliziert werden, denn nachdem der fehlerhafte Teil „gefixed“ wurde, können die Veränderungen, in diesem Teil, andere Fehler verursachen [2].

Im Allgemeinen gilt also, dass die Fehlerlokalisierung der zeitintensive und das Bugfixing der komplizierte Teil ist. Um den Aufwand gering zu halten, sollte möglichst früh, also in den ersten Phasen schon, mit dem Schreiben der Tests begonnen werden. Selbst kleinere Teile eines Programms sollten früh genug getestet werden, da in den späteren Phasen das Programm sehr komplex wird und somit auch die Fehleruntersuchung schwieriger wird [1].

3 Speicherfehler in C

In der Programmiersprache C erfolgt die Speicherverwaltung nicht, wie in anderen Sprachen (Java), automatisch. Daher können häufig Fehler in der Speicherverwaltung auftreten.
Einige der typischen Speicherfehler werden in folgendem Abschnitt näher erläutert, damit diese Fehler dem Programmierer bekannt sind und vermieden werden können.

3.1 Nicht initialisierte Variablen

In der Programmiersprache C muss eine Variable nicht zwingend initialisiert werden. Auch ohne die Initialisierung kann der Compiler übersetzen.
So kann es beispielsweise passieren, dass der Programmierer entweder die Initialisierung einer lokalen Variable vergisst oder sogar davon ausgeht, dass ohne die Initialisierung der Wert 0 zugewiesen wird. Jedoch bekommt eine nicht initialisierte lokale Variable einen unbekannten beliebigen Wert. Durch solche Werte kann es dazu kommen, dass sich das Programm nicht mehr korrekt verhält, da falsche Werte übergeben wurden und falsche Ergebnisse zurückgeliefert werden können [3].

Vermieden kann dieser typische Fehler, indem die Initialisierung zur Angewohnheit wird und die Variablen immer automatisch von dem Programmierer initialisiert werden. Da aber auch Programmierer Fehler machen können und die Initialisierung vergessen können, gibt es zusätzlich auch Warnungen vom Compiler, die fehlende Initialisierungen aufzeigen.

3.2 Der Gültigkeitsbereich

Es geht im Folgenden um die Erreichbarkeit und das Verschatten der Variablen. Variablen sind innerhalb ihres Anweisungsblockes erreichbar. Die Globalen Variablen werden außerhalb von allen Anweisungsblöcken definiert und sind somit in allen Anweisungsblöcken verfügbar.

Die Verfügbarkeit von Variablen kann eingeschränkt werden, indem verschattet wird. Verschatten bedeutet, dass man die gleiche Bezeichnung, die schon für eine globale Variable verwendet wird, auch für eine lokale Variable benutzt. Innerhalb des Anweisungsblockes dieser lokalen Variable, ist nicht mehr die globale, sondern nur die lokale Variable erreichbar.

Bewusst verschatten verursacht keinen Fehler. Erst wenn der Programmierer den Überblick über die schon verwendeten Bezeichner verliert und unabsichtlich verschattet, wird das Programm nicht mehr korrekt laufen. Es können von den falschen Variablen Werte ausgelesen und falsche Variablen überschrieben werden.

Verhindert kann dies durch die Wahl von Variablenbezeichnungen, die verständlich sind. Bezeichner wie „x“ oder „y“ sind ungünstig.
Compiler können dies erkennen und geben Warnungen aus, die ernst genommen werden sollten.

3.3 Null-Pointer

Wenn eine Allocation, also Reservierung vom Speicher, fehlschlägt, bekommen Pointer den Wert des Null-Pointers.
Nach einem versuchten Zugriff auf einen Null-Pointer, bricht das Betriebssystem die Ausführung des Programms ab. Hierbei handelt es sich um einen Seitenfehler. Es wird versucht auf einen Speicher zuzugreifen, welcher nicht für dieses Programm zugelassen ist [4].

3.4 Allocation

Allocation bedeutet das Reservieren vom Speicher. Es gibt verschiedene Varianten, wie Speicher reseviert werden kann.

3.4.1 Memory Allocation: „void *malloc(size_t size);”

Mit Memory Allocation (malloc) wird zur Laufzeit Speicherplatz reserviert. Mit dem Parameter „size“ wird die Größe, die benötigt wird, in Byte übergeben. Der Rückgabewert ist ein void-Zeiger oder ein null-Zeiger, falls kein freier Speicher mehr zur Verfügung steht [5].

3.4.2 Cleared Memory Allocation: „void *calloc(size_t n, size_t size);“

Mit Cleared Memory Allocation (calloc) wird auch Speicher reserviert. Mit Memory Allocation wurde nach Bytes reserviert, mit „calloc“ wird nach Elementen reserviert. „Calloc“ und „malloc“ unterscheiden sich dadurch, dass calloc die Speicherstellen mit 0 initialisiert, wobei „malloc“ den Speicherinhalt unverändert lässt. Die Rückgabewerte sind im Endeffekt identisch [5].

3.5 Speicherlecks

Sobald ein Speicher alloziiert wurde und nicht mehr benötigt wird, ist es wichtig, dass es sofort wieder freigegeben wird. Wird regelmäßig Speicher reserviert, ohne ihn wieder freizugeben, kann es passieren, dass das System zum Erliegen kommt. Die unnötig reservierten und nicht wieder freigegebenen Speicherbereiche bleiben über die gesamte Laufzeit des Programms für das System unbrauchbar. Es kann also dazu kommen, dass der Speicher komplett belegt ist und die Daten auf die Festplatte ausgelagert werden müssen. Dieser Prozess kann relativ lange dauern [6].

Mit „free“ kann der Speicher freigegeben werden, der mit malloc oder calloc reserviert wurde.
Zudem darf der Speicher, der mit „free“ freizugeben ist, nicht vorher freigegeben sein.

Im folgenden Beispiel ist zu sehen, wie für die Variable „test“ zuerst Speicher reserviert und wie dieser Speicher wieder freigegeben wird.

```c
#include<stdio.h>
#include<stdlib.h>

int main() {
    int size=100;
    int *test;

    // Speicher reservieren
    test = (int *) calloc(size, sizeof(int));

    // Speicher freigeben
    free(test);

    return 0;
}
```

3.6 Zugriff nach Freigabe

Nachdem ein Speicher mit „free“ freigegeben wurde, besteht die Gefahr, dass trotzdem noch darauf zugegriffen wird.
Dies stellt deswegen eine Gefahr dar, da dieser Speicher nicht mehr zur Verfügung steht und schon für andere Dinge in Verwendung sein kann. Zudem wird das Programm nicht unbedingt abstürzen, da der Speicher möglicherweise immer noch dem Programm gehört [7].

3.7 Doppelte Freigabe

Wenn der Programmierer vergisst, dass schon ein Speicher freigegeben wurde und es noch mal für denselben Speicher tut, entstehen unerwünschte Reaktionen. Das Resultat ist undefiniert [7].

In folgendem Beispiel ist zu sehen, wie ein Speicher freigegeben wird und versehentlich noch mal versucht wird, diesen wieder freizugeben.

```
int *z;
int *z2;

//Speicher wird hier reserviert
z = malloc(sizeof(*));

z2 = z;

// z wird freigegeben
free(z);

// Fehler: ist schon freigegeben, da z schon
// freigegeben wurde.
free(z2);
```

4 Sanitizer

Durch die Speicherfehler entstehen viele Sicherheitslücken. Daher ist es sehr wichtig diese Fehler zu finden. Es gibt verschiedene Ansätze, die Speicherfehler finden oder keine Speicherfehler erlauben. „Das Projekt „Softbound + CETS“ baut durch eine Erweiterung des Clang-Compilers von LLVM zusätzliche Checks in den generierten Code ein, um Zugriffe außerhalb zulässiger Speicherbereiche zu unterbinden“ Dieser Ansatz funktioniert zwar, jedoch sind die Performanceverluste hoch und lägen bei über 100 Prozent[8].

4.1 AddressSanitizer

Der AddressSanitizer ist ein Hilfsmittel um Speicherfehler zu finden [10]. Er ist ein Teil von Gcc und Clang. Während der Übersetzungszeit besteht die Möglichkeit den Code zu instrumentieren, damit dann zur Laufzeit Speicherfehler gefunden werden können. Fehler wie Use-after-free, Double-free, Memory-leak, etc. können gefunden werden. Der Speicherverbrauch sei bis zu dreimal so groß und der Performanceverlust läge etwa zwischen 70 und 100 Prozent. Dieser Ansatz ist weniger aufwändig als der vorherige, trotzdem ist der Performanceverlust zu hoch, da die Ausführungszeit sich ungefähr verdoppelt. Daher wird der AddressSanitizer hauptsächlich als Debugging-Tool genutzt[8].

4.2 MemorySanitizer

Der Memory Sanitizer ist in Clang integriert. Dieser kann Speicherfehler wie nicht initialisierte Variablen finden [11]. Auch dieser Sanitizer besteht aus einer Instrumentationskomponente, die den Code während der Übersetzungszeit instrumentiert und einer Laufzeitkomponente, die den Code zur Laufzeit auswertet und die speziellen Speicherfehler findet.
Hier können sich die Ausführungszeit und der Speicherverbrauch zur Laufzeit bis zu verdreifachen[9].

4.3 LeakSanitizer

Der Leak Sanitizer kann zur Laufzeit Speicherlecks finden. Dieser Sanitizer kann auch mit dem Address Sanitizer kombiniert werden. Auch hier gilt aber, dass die Perfomance stark beeinträchtigt wird [12].

Zusammenfassend finden die Sanitizer viele der Speicherfehler. Der Memory Sanitizer jedoch sei jedoch noch in einem experimentellen Zustand und habe noch einige Fehler. Trotzdem sind die Sanitizer im Allgemeinen eine gute Hilfe um Speicherfehler zu finden. Der Nachteil ist jedoch, dass sie nicht effizient sind. Die Ausführungszeit und der Speicherverbrauch verdoppeln sich oder verdreifachen sich sogar.

5 Effizientes Fehlerfinden

5.1 Suchraum einschränken

Damit nicht der gesamte Quellcode auf Fehler untersucht werden muss, sondern nur die Stellen in denen Änderungen vorgenommen wurden, werden Methoden wie Delta Debugging eingesetzt.

Nehmen wir an, dass zwei Versionen eines Projektes mit entsprechenden funktionalen Tests vorliegen. Die Tests der älteren Version sind erfolgreich und einige Tests der neueren Version schlagen fehl. Dann kann die Abweichung beider Versionen, also das „Delta“ bestimmt werden [13].

So stellt sich heraus welche der Anweisungen von „Delta“ dafür verantwortlich sind, dass dieser Test fehlschlägt.
Mit diesem Ansatz kann die zu untersuchenden Stellen eingeschränkt werden.

5.2 Delta Debugging

Delta Debugging ist ein automatisches Verfahren, das die fehlerverursachenden Änderungen findet. Dabei wird schrittweise versucht, Regionen auszuschließen, die nicht für den Fehler verantwortlich sind [13].
In Folgendem wird Delta Debugging anhand eines Beispiels näher erläutert.

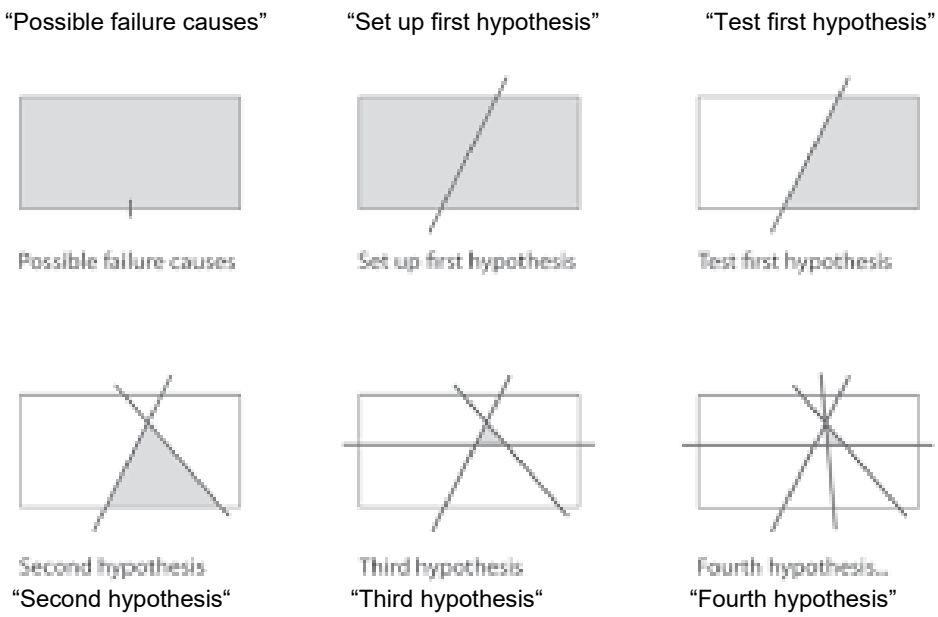

Abbildung 1: Das Prinzip des Delta Debuggings graphisch dargestellt. Durch das ständige Wiederholen der Iteration wird der zu suchende Raum kleiner. [Universität Saarland. http://www.st.cs.uni-sb.de/dd/.]

5.2.1 Beispiel 1

Schritt	C	Konfiguration	Test
1	C1	1 2 3 4	+
2	C2	. . . . 5 6 7 8	+
3	C3	1 2 . . 5 6 7 8	+
4	C4	. . 3 4 5 6 7 8	-
5	C5	. . . 4 5 6 7 8	+
6	C6	. . 3 . 5 6 7 8	-
7	C7	1 2 3 4 . . 7 8	+
8	C8	1 2 3 4 5 6 . .	-
9	C9	1 2 3 4 5 . . .	+
10	C10	1 2 3 4 . 6 . .	-
Ergebnis		. . 3 . . 6 . .	

An diesem Beispiel wird die Funktionsweise des Delta Debuggers demonstriert. Die Mengen aller Änderungen sind durch die Zahlen von 1 bis 8 dargestellt. Zudem befindet sich in dem Beispiel eine c-Spalte, welche die Bezeichnungen für die Konfigurationen beinhaltet. Die Schritte 1 und 2 testen zwei Hälften (C_1 und C_2) der gesamten Änderungsmenge. Beide bestehen den Test. Dies bedeutet, dass der Fehler nur durch eine Kombination von Änderungen aus den Konfigurationen C_1 und C_2 hervorgerufen wird. Dies bedeutet, dass eine Interferenz vorliegen muss. Deshalb muss einer der beiden Teile fest sein und mit der anderen Hälfte der Algorithmus neugestartet werden. In diesem Fall ist C_2 fest. Nach Schritt 4 wird deutlich, dass der Fehler sich in dieser Änderungsmenge(3…8) befindet. So wird mit dieser Menge in Schritt 5 weiterverfahren. Die Änderungen 3 und 4 werden einzeln mit C_2 getestet und es stellt sich heraus, dass in der ersten Hälfte die Änderung 3 mit einer Änderung aus C_2 einen Fehler verursacht. Nun wird die Änderung in der zweiten Hälfe gesucht, die in Kombination mit der Änderung 3 den Fehler verursacht. Dafür wird die erste Hälfte festgehalten und mit der zweiten Hälfte weiterverfahren. In Schritt 8 bestehen die Änderungen 5 und 6 den Test nicht. So muss mit diesen beiden Änderungen weiterverfahren werden. Schließlich besteht in Schritt 10 die Änderung 6 nicht den Test und verursacht somit in Kombination mit der Änderung 3 einen Fehler.

Mit diesem Beispiel wird unterstrichen, dass Delta Debugging ein fehlerlokalisierender Algorithmus ist. Die Änderungen, die nicht geändert wurden oder die die Tests bestanden haben, müssen nicht weiter berücksichtigt werden. Somit wird die Menge der Änderungen, die möglicherweise einen Fehler verursachen, kleiner.

5.3 Zeitnahes Fehlerfinden (Continuous Testing)

In Entwicklungsumgebungen wie Eclipse findet die Kompilierung im Hintergrund statt. Während Funktionen oder sonstige Modifikationen am Quelltext vorgenommen werden, wird ständig, ohne dass explizit aufgefordert wird, der modifizierte Programmteil kompiliert. Dies bedeutet, dass dem Programmierer sofort die Programmfehler, die der Compiler erkennen kann, sichtbar gemacht werden. Dadurch erhöht sich auch die Qualität, da der Programmierer während der Programmierung schon auf Fehler und Warnungen aufmerksam gemacht wird. Somit ist auch der Prozess des Fehlerbehebens verkürzt, da der Programmierer während der Modifikation darauf reagieren kann, denn die logische Struktur des Problems muss nicht erneut gedanklich aufgebaut werden. Die Navigation zum Auftrittsort des Programmfehlers entfällt auch.

Die Tests im Programm können manuell ausgeführt werden oder sie laufen ständig asynchron im Hintergrund. Die aktuelle Version des Quellcodes wird mit der letzten erfolgreichen Testdurchführung verglichen, damit nur die Tests wiederholt werden, die von einer Modifikation betroffen sind. Tests können auch automatisch priorisiert werden. Der Most-Recent-Failure-First-Ansatz hat sich als nützlich herausgestellt. Wenn ein Test aktuell fehlgeschlagen oder vor kurzem fehlgeschlagen ist, besteht die Gefahr, dass er wieder fehlschlägt. Deshalb sind solche Tests interessant für den Programmierer. Zusammenfassend ist Continuous Testing ein Plug-In, das Fehler vor allem zeitnah aufdeckt und die Tests stetig durchführt [14].

6 Valgrind

Mit dem Werkzeug Valgrind werden jegliche Arten von Speicherzugriffen zur Laufzeit überprüft. Dabei simuliert Valgrind einen kompletten Prozessor [15]. Er liest Binaries mit Debuginformationen ein, damit dann bei Fehlern auf die korrekte Quellcodezeile verwiesen werden kann. Diese werden in ein Zwischenformat übersetzt und Überprüfungsanweisungen in dieser Sprache hinzugefügt. Anschließend wird es dann wieder nach Maschinencode umgewandelt. Diese Aufwendigkeit macht sich auch bei der Performance deutlich bemerkbar. Die Ausführungszeit wächst auf ungefähr das 10 bis 50 fache an [2]. Dieser Performanceverlust kann nur dann akzeptabel sein, wenn es zu Testzwecken benutzt wird.

7 Zusammenfassung

Die Sprache C ist wegen der manuellen Speicherverwaltung sehr beliebt. Dadurch kann Performance eines Programmes gesteigert werden. Jedoch können sehr viele Speicherfehler entstehen, die nicht schnell und leicht zu finden sind. Durch die Speicherfehler können Sicherheitslücken entstehen. Daher ist es wichtig diese Fehler zu finden und zu beheben. Das Werkzeug Valgrind kann zwar jegliche Arten von Speicherfehler finden, lässt aber die Ausführungszeit auf das 10 bis 50 fache wachsen. Eine Alternative sind die Sanitizer wie AddressSanitizer, LeakSanitizer, MemorySanitizer etc..
Diese können kombiniert werden und finden auch viele Arten der Speicherfehler. Bei den Sanitizer verdoppeln bis verdreifachen sich der Speicherverbrauch und die Ausführungszeit und sind somit effizienter als Valgrind. Trotzdem ist es ein starker Performanceverlust.

Um nicht nur Speicherfehler, sondern auch andere Fehler finden zu können, gibt es Ansätze wie Delta Debugging und Continuous Testing. Delta Debugging und Continuous Testing schließen sich gegenseitig nicht aus. Denn Continuous Testing führt asymmetrisch die Tests im Hintergrund durch und findet die Fehler vor allem zeitnah.
Delta Debugging verkleinert die bestehende Menge an fehlerverursachenden Anweisungen, indem der Suchraum mit dem Versuch- und Irrtum Prinzip eingegrenzt wird.

8 Literatur

[1] Bugs und Fehlersuche, https://codingtutor.de/debugging/, 03.03.2017

[2] Debugging, www2.in.tum.de/hp/file?fid=1239, 03.03.2017

[3] Nicht initialisierte Variablen, https://en.wikipedia.org/wiki/Uninitialized_variable, 03.03.2017

[4] Pointer, https://www.luis.uni-hannover.de/fileadmin/kurse/material/CKurs/c6_Zeiger.pdf, 03.03.2017

[5] Allocation, http://www.c-howto.de/tutorial-arrays-felder-speicherverwaltung.html, 03.03.2017

[6] Memory leaks, https://en.wikipedia.org/wiki/Memory_leak, 03.03.2017

[7] Speicherverwaltung, https://de.wikibooks.org/wiki/CProgrammierung:_Speicherverwaltung 03.03.2017

[8] Pointer, https://www.golem.de/news/c-programmierung-schutz-fuer-code-pointer-1412-111364.html, 03.03.2017

[9] Vergleich unterschiedlicher Compiler,ftp://ftp.informatik.uni-stuttgart.de/pub/library/medoc.ustuttgart_fi/FACH-0198/FACH-0198.pdf, 03.03.2017

[10] AddressSanitizer, http://clang.llvm.org/docs/AddressSanitizer.html, 03.03.2017

[11] MemorySanitizer, http://clang.llvm.org/docs/MemorySanitizer.html, 03.03.2017

[12] LeakSanitizer, http://clang.llvm.org/docs/LeakSanitizer.html, 03.03.2017

[13] Fehlerlokalisierung, https://www.fernuni-hagen.de/ps/arbeiten/master_bertschler.shtml, 03.03.2017

[14] Martin Fowler. "Continuous Integration". http://martinfowler.com/articles/continuousIntegration.html, 03.03.2017

[15] Valgrind, http://valgrind.org/docs/memcheck2005.pdf, 03.03.2017