Typische Speicherfehler in C

Thorsten Ploß
Informatik Universität Hamburg
Proseminar: C-Grundlagen und Konzepte

15. Juli 2013

Inhaltsverzeichnis

1 Einleitung

Dank der Optimierung auf Geschwindigkeit und Assamblernähe ist C eine beliebte, weit verbreitete Programmiersprache, welche in vielen Gebieten Einsatz findet.
Diese Optimierung fordert jedoch, dass auf einigen Luxus, der aus anderen Hochsprachen bekannt ist, verzichtet werden muss.
Daher ist es für jeden, der in C programmieren will essentiell zu wissen, worauf beim Programmieren zu achten ist, worum sich in anderen Sprachen der Computer kümmert und welche Gefahren davon ausgehen, nicht sorgfältig zu arbeiten.
Daher werden hier ein paar der häufigsten Fehler, die in der Speicherverwaltung auftreten, betrachtet.

2 Klassische Speicherverwaltung - Fehlerquellen

Auch ohne manuelle Speicherverwaltung hat C einige Eigenschaften, die zu Fehlern führen können. Von diesen sollten zumindest ein paar näher betrachtet werden.

2.1 Nicht initialisierte Variablen

Die C-Standards definieren nur für globale und statische Variablen, welchen Wert Variablen nach ihrer Definition haben.
Diese werden standardmäßig mit dem Wert 0 initialisiert.
Weiterhin definieren sie nicht, dass Variablen initialisiert werden müssen.
Daraus folgt, dass man bei einer lokalen Variable das initialisieren vergessen oder davon ausgehend, dass ihr Wert 0 sein wird, weglassen kann.
Ohne Initialisierung haben diese Variablen unbekannte, beliebige Werte, welche ein undefiniertes Verhalten des Programms nach sich zieht.
Dies lässt sich vermeiden, indem man sich generell die Initialisierung per Hand angewöhnt und per hohem Warninglevel des Compilers fehlende Initialisierungen findet und behebt.

2.2 Der Gültigkeitsbereich

Der Gültigkeitsbereich in C ist über Anweisungsblöcke definiert. Diese sind durch ihre Klammerung mit geschweiften Klammern zu erkennen.
Jede Variable, die definiert wird, ist in jedem Anweisungsblock definiert, welcher sich innerhalb des gleichen Anweisungsblocks befindet.
Globale Variablen werden außerhalb von allen Anweisungsblöcken definiert und sind daher in allen Anweisungsblöcken verfügbar.
Für als static definierte Variablen gilt das gleiche, wie für globale Variablen, außer, dass sie nicht in Methoden verwendet werden können, welche sich vor der Definition der Variable befinden.

Die Verfügbarkeit von Variablen kann man jedoch einschränken, indem man sie verschattet. Verschatten bedeutet, einen Bezeichner für eine Variable, welche aus einem höher liegenden Anweisungsblock stammt, für einen neue Variable zu verwenden. Innerhalb dieses Anweisungsblocks ist damit nur noch die neue Variable verfügbar, nicht jedoch die weiter außen definierte.
Da Variablen am Ende des Anweisungsblocks, in dem sie definiert werden, standardmäßig verfallen, ist die ältere Variable nach Ende des Anweisungsblocks wieder erreichbar.

Der Versuch, in einem Programm eine Variable außerhalb ihres Gültigkeitsbereichs nutzen zu wollen, scheitert, wenn es keine andere Variable mit diesem Bezeichner gibt, beim Kompilieren.
Daher stellt dies keine Gefahr dar, sondern nur Aufwand bei der Fehlersuche.

Die Verschattung führt eher zu Problemen, in Fällen, in denen man den Überblick über schon verwendete Bezeichner verliert und daher unabsichtlich verschattet. Dadurch können unabsichtlich die falschen Variablen verändert werden, wodurch Änderungen verloren gehen.
Des weiteren bietet C die Möglichkeit Variablen zu verschatten, welche von includierten Programmteilen genutzt werden, wodurch unübersichtliche Fehler entstehen können, welche nur sehr schwer aufzudecken sind.

Dagegen wirken kann man prinzipiell durch eine gute Wahl an Bezeichnern.
Diese sollten immer aufschlussreich sein und damit auch eine brauchbare Länge haben.
Bezeichner, die aus einem Buchstaben bestehen, neigen zur Endlichkeit, sowie zur geringen Verständlichkeit.

2.3 Das Ende des Arrays

Der Zugriff auf ein Array ist in C nicht gesichert.
Es ist valide auf Indizes zuzugreifen, die den größten Index des Array überschreiten oder gar auf negative Indizes zuzugreifen.
Da auf diese Weise Zugriffe auf Speicher gewährt werden, welcher auf unbekannte Weise genutzt wird, ist dies ein großes Risiko.

Da Char-Arrays verwendet werden um Strings darzustellen und diese oft in Benutzereingaben Verwendung finden, bieten interaktive Programme ein noch weitaus größeres Risiko. Werden hier keine gesonderten Schutzmaßnamen getroffen, bieten String-Eingaben Angreifern die Möglichkeit große Speicherbereiche zu verändern.
Für den Angreifer im Idealfall erfährt er, wie er Zugriff auf die Rücksprungadresse des Programms nehmen kann, wodurch es ihm möglich wird, das Programm zum Ausführen beliebigen Codes zu bringen.

Da Fehler dieser Art vom Compiler nicht erkannt werden können und erst verspätet oder gar keinen Einfluss auf die Funktion des Programms haben, sind diese Fehler äußerst schwer zu finden.
Umso wichtiger ist die Vorbeugung. Berechnet man, auf welchen Index man zugreifen möchte, ist es empfehlenswert eine Abfrage zu machen, ob dieser Index sich tatsächlich innerhalb der Grenzen des Arrays befindet, bevor ein Zugriff durchgeführt wird.
Eine sichere Benutzereingabe kann man z.B. mit der Methode fgets() aus dem Header stdio.h erreichen, denn diese ließt nur eine bestimmte, wählbare Länge an Zeichen ein.

3 Dynamische Speicherverwaltung - Fehlerquellen

Das mächtige Werkzeug der dynamischen Speicherverwaltung bietet viele Vorteile. Jedoch bietet dieses auch eine Sammlung an Fehlerquellen, von denen die häufigsten hier Behandlung finden. Dabei werden grundlegende Kenntnisse über die dynamische Speicherverwaltung voraus gesetzt.
Informationen über die dynamische Speicherverwaltung finden Sie unter anderem hier: http://wr.informatik.uni-hamburg.de/_media/teaching/sommersemester_2013/cgk-13-dobert-ausarbeitung.pdf

3.1 Null-Pointer

Laut Standard haben globale Pointer nach der Definiton den Wert des Null-Pointers.
Darüber hinaus haben Pointer diesen Wert wenn eine Allocation fehlschlägt.
In den meisten Fällen bildet dies keine Gefahren, da das Betriebssystem ein Programm terminiert, welches einen Zugriff auf einen Null-Pointer versucht.
Dieser Versuch entspricht einem Seitenfehler, einem Zugriff auf Speicher, welcher nicht für dieses Programm zugelassen ist.
Für diesen Fehler sind auch die englischen Begriffe "pagefault" oder "segfault" beliebt.

Allerdings gibt es auch Situationen, in denen kein Betriebssystem zur Verfügung steht, um den Zugriff auf den Null-Pointer zu vermeiden.
In Bereichen der Microkontroller-Programmierung oder der Programmierung von Betriebssystemen fehlt ein Schutz vor Zugriffen auf den Null-Pointer, wodurch für das System wichtige Speicherbereiche betroffen sein würden.

Um unerwünschte Programmabstürze zu vermeiden, sollte nach eine Allocation immer überprüft werden, ob diese gescheitert ist.
Falls man tatsächlich in den seltenen Fall kommt, dass Null-Pointer für das

System gefährlich werden könnten, ist generell ein hohes Maß an Vorsicht empfohlen.

3.2 Pointerprobleme

Der Umgang mit Pointern hat einige Eigenheiten, welche Risiken bergen.

3.2.1 Speicherallocation

Die Allocation, die Reservierung von Speicher auf dem Heap, ist der wichtigste Vorgang bei der dynamischen Speicherverwaltung.
Man bittet damit das Betriebssystem darum, einen Speicherbereich bestimmter Größe zu reservieren und bekommt die Adresse, mit der dieser Speicherbereich beginnt zurück, welche man dann einem Pointer zuweist.
Da man angeben muss, wie groß der Speicherbereich sein soll, bietet sich die Gelegenheit, sich zu vertun.

Da einige Datentypen auf verschiedenen Betriebssystemen unterschiedlich viel Speicher benötigen, ist eine hart im Code stehende Größenangabe für den zu allozierenden Speicher eine Gefahr für die Verwendbarkeit eines Programms auf verschiedenen Betriebssystemen.
Im schlimmsten Fall alloziert man zu wenig Speicher und überschreibt ungewollt, unbekannte Speicheradressen.

Hiergegen hilft effektiv die Methode "sizeof()".
Damit kann man sich von dem Datentyp, den man braucht, die Größe geben lassen, um immer die richtige Menge Speicher zu allozieren.

3.2.2 Syntaxverwechslung

Durch die unterschiedlichen Zeichen, welche vor den Bezeichner geschrieben werden müssen, um nun auf den Speicher zuzugreifen, der im Pointer gespeichert wird, auf die Adresse, die dieser Speicher hat oder auf die Adresse, die auf der der Pointer gespeichert ist zu referenzieren entstehen leicht kleine Verwechslungen der Syntax, welche schwerwiegende Einflüsse auf das Programm haben.

```
1  int main(int argc, char *argv[])
2  {
3      char *string=NULL;
4      char[] string2 = "Otto";
5
6      string = malloc(sizeof(char)*6); //Zuweisung einer Speicheradresse
7
8      *string = "Hallo"; //Speicherung eines Wertes in die Speicheradresse
9
10     string = &string2; //Zuweisung einer Speicheradresse auf eine Adresse,
11     //die von einer anderen Variable genutzt wird.
12 }
```

Ohne Zusatz erhält man durch den Bezeichner eines Pointers die Adresse, auf die er zeigt.
Mit Hilfe eines vorangestellten "*" erhält man den Wert, der in der Adresse gespeichert ist.
Mit Hilfe eines vorangestellten "&" erhält man die Adresse, in der der Pointer oder eine andere Variable gespeichert ist.

Ein Verwechseln dieser führt im besten Fall zu Fehlermeldungen, im schlimmsten Fall zu undefiniertem Programmverhalten.

3.3 Speicherlecks

Zu einer vollständigen dynamischen Speicherverwaltung gehört nicht nur das Reservieren und das Nutzen von Speicher auf dem Heap, sondern auch das Freigeben dessen.
Dazu gibt es die Operation "free()", mit der man allozierten Speicher wieder frei geben kann.
Vergisst man dies, bleibt über die gesamte Laufzeit des Programms dieser Speicherbereich reserviert und damit auch noch lang nachdem er nutzlos geworden ist unbrauchbar für den Rest des Systems.
Geschieht dies wiederholt, kann man sich den Speicher wie einen Tank gefüllt mit Treibstoff vorstellen, welcher leck schlägt.
Es ist immer weniger freier Speicher vorhanden, bis das System zum Erliegen kommt.
Man spricht von einem Speicherleck.
Da dieser Vorgang sehr langsam sein kann, die Auswirkungen weitreichend sind und in größeren Projekten nur sehr schwer lokalisierbar sind, ist dieser Fehler gefürchtet.

Vorsicht ist die beste Möglichkeit, diesem Fehler vorzubeugen.
Zur Aufdeckung von Fehlern dieser Art gibt es eine Sammlung an Tools, welche allerdings nur Vermutungen anstellen können, dass es diesen Fehler gibt, da ein wachsender Speicherbedarf auch beabsichtigt sein kann.

3.4 Benutzung nach Freigabe und doppelte Freigabe

Die Freigabe bietet aber auch Gefahren.
So bleibt auch nach der Freigabe der Pointer auf die Adresse unverändert, sodass man weiterhin darauf zugreifen kann.
Da jedoch der Speicher für andere Zwecke schon wieder in Verwendung sein kann, bieten solche Zugriffe Gefahren.
Wenn man nicht oder nicht mehr weiß, dass ein Speicher bereits freigegeben wurde, kann es auch passieren, dass man ihn ein zweites mal freizugeben versucht.
Was daraus resultiert ist nicht definiert und damit auch nicht erwünscht.

Gängige Praxis diese Fehler zu vermeiden, ist es dem Pointer nach seiner Freigabe die Adresse des Null-Pointers zuzuweisen.
Weitere Zugriffsversuche enden damit mit einem Segfault und sind daher leicht gefunden.
Eine Freigabe eines Null-Pointers bleibt, laut C Standards, ohne Folgen, stellt also auch keine Gefahr mehr da.

4 Resumee

Diese Sammlung an Fehlerquellen zeigt eindeutig auf, dass die Verwendung von C nur mit gewissenhafter Vermeidung von Fehlern zum Erfolg führt und damit nicht geeignet ist für Kurzschlussprojekte. Ohne das Vorwissen um die Eigenschaften der Programmiersprache entstehen sehr leicht kleine Fehler mir großen Wirkungen.
Die Verwendung von C ist damit nur zu empfehlen, wenn man informiert ist über die Tücken und den Aufwand nicht scheut, sie zu umgehen.

5 Quellen

- http://www.peace-software.de/ckurs12.html
  [Stand: 13.05.2013]
- http://pronix.linuxdelta.de/C/standard_C/
  c_programmierung_17.shtml [Stand: 13.05.2013]
- http://de.wikibooks.org/wiki/C-Programmierung:_Speicherverwaltung
  [Stand: 13.05.2013]
- https://de.wikipedia.org/wiki/Speicherleck
  [Stand: 13.05.2013]
- http://c-buch.sommergut.de/Kapitel10/G%FCltigkeitsbereich-von-Variablen.shtml
  [Stand: 13.05.2013]
- http://wr.informatik.uni-hamburg.de/_media/teaching/sommersemester_2013/cgk-13-dobert-ausarbeitung.pdf
  [Stand: 11.07.2013]