Universität Hamburg

Seminar: Effiziente Programmierung in C

# Referenzzählung

*Autoren:*
Kevin Köster

*Betreuer:*
Michael Kuhn

31. März 2013

# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Das Problem

In einem Programm fragt man sich oft nicht, wem nun ein Datenblock gehört, da es oft eindeutig ist. Jedoch gibt es auch viele Situationen in denen nicht klar ist, ob ein Datenbereich noch benötigt wird oder nicht.
Wird der Datenbereich nun nicht freigegeben, könnte es zu einem Speicherleck (memory leak) kommen, wodurch das Programm unnötig viel Speicher verbraucht und möglicherweise irgendwann vom Betriebssystem beendet wird.
Es könnte sich hierbei zum Beispiel um ein Icon handeln, welches temporär an mehreren Stellen im Programm benötigt wird. Somit weiß man nicht, ob das Icon noch irgendwo anders benötigt wird oder nicht. Hier kann man entweder aktiv nachfragen oder einfach die Referenzzählung benutzen.

## 1.2 Was ist Referenzzählung?

Bei der Referenzzählung (engl. Reference Counting) handelt es sich um ein Verfahren das Aufkommen bestimmter Daten zu zählen.
Dabei wird bei jedem zu zählendem Wert eine Variable gespeichert, welche die Anzahl der Referenzen zählt.
Fällt dieser Zähler auf 0, wird das Objekt freigegeben.

## 1.3 Beispiel: Dateisystem

An dem Beispiel des Dateisystems ext2 möchte hier die Notwendigkeit der Referenzzählung darstellen.

**Hinweis**

Es sollte auch alles ohne weiteres auf die Dateisysteme ext3 und ext4 übertragbar sein. Jedoch könnte es minimale Unterschiede geben.

Bei diesem Dateisystem werden die Dateien über Index Nodes (Inodes) verwaltet. Diese enthalten Metainformationen über die Datei. Ein Inode hat dabei eine eindeutige Nummer und steht mit den Dateien in einer 1:1 Beziehung. Also verweist ein Inode auf genau eine Datei und auf eine Datei zeigt genau ein Inode.
Die Metatadaten eines Inodes enthält unter anderem die Zugriffsrechte der Datei, die Eigentümer, der Dateityp, Größe der Datei, Verweise auf die Blöcke, wo die Datei liegt, Zeiten der letzten Zugriffe und Änderungen, sowie einen Referenzzähler. Für uns interessant ist hier der Referenzzähler. Dieser gibt an, wie viele Verweise es auf diesen Inode gibt.
Jeder Dateiname (genauer Dateipfad) zeigt dabei auf einen bestimmtem Inode. Diese Information wird wiederum an einem anderen Ort gespeichert.
Durch diese Eigenschaften ist es möglich dass mehrere Dateipfade auf den gleichen Inode und somit auf die gleiche Datei verweisen.

**Warnung**

Die 1:1 Beziehung gilt nur für Inodes und Daten. Nicht für Dateipfade und Inodes.

Somit kann man jede Datei mit einem Pointer in C vergleichen, welcher auf einen Datenblock zeigt.
Jede Datei ist somit ein sogenannter Hardlink. Da man hier nun beim Löschen einer Datei nicht wissen kann, ob noch ein weiter Verweis auf den Inode existiert, zählt man einfach die Referenzen auf diesen Inode.
Wird ein weiterer Harlink erzeugt, erhöht sich der Zähler.
Dadurch ist das Löschen einer Datei auch nicht immer ein wirkliches Löschen. Es wird lediglich der Verzeichniseintrag entfernt und der Referenzzähler wird dekrementiert. Nur wenn dieser auf 0 fällt, werden die Daten gelöscht.
Dies ist ein deutliches Beispiel, wo Referenzzählung erforderlich ist und andere Lösungen viel umständlicher oder gar unmöglich wären.

## 1.4 Weitere Anwendungen

Zudem findet man in fast jeder Scriptsprache die Referenzzählung. In den meisten Sprachen wird jede Variable (bzw. deren Inhalt) mit einem Zähler versehen.
So wird zum Beispiel in Python jeder String oder andere Variable nur einmal gespeichert und die Referenzen auf diesen gespeichert.

```
1 #!/usr/bin/env python
2 import sys
3
4 a = 42
5 print(sys.getrefcount(a))
6 b = a
7 print(sys.getrefcount(a))
```

Listing 1: Python Referenz

Führt man dieses Programm aus, erhöht sich die Referenz jeweils um 1. Dabei kann es vorkommen, dass die erste Ausgabe zum Beispiel bereits 9 ist. Dies liegt daran, dass der Wert der Variable bereits irgendwo referenziert wird.
Auch im weit verbreitetem PHP wird die Referenzzählung für die Variablen verwendet.

## 1.5 Probleme

Natürlich kann es bei der Referenzzählung auch zu Problemen kommen. Zum einen ist es natürlich etwas mehr Aufwand Daten mit einem Zähler zu speichern. Zum einen Verbraucht man etwas mehr Speicher und zum anderen benötigt man etwas mehr Zeit, da man eventuell mit Pointern rechnen muss und auf Speicher zugreifen muss.
Zum anderen kann man durch die Referenzzählung auch Speicherlecks erzeugen. Dies kann durch sogenannte Zyklen passieren.
Bei den Zyklen zeigen 2 oder mehr Speicherbereichen in einer zyklischen Beziehung

aufeinander. Entfernt man nun alle anderen Referenzen, bleibt nur noch der Zyklus übrig. Da in diesen Zyklus jedes Objekt referenziert wird, kann der Referenzzähler nicht auf 0 fallen und der Speicher wird somit nicht freigegeben und ist nicht mehr von außen Zugreifbar. Der Speicher ist somit "verloren".
Verhindern kann man diese Zyklen, indem man sogenannte weak Pointer verwendet. Diese Zeigen auf einen Speicherbereich, erhöhen aber den Referenzzähler nicht. Möchte man diesen benutzen, kann man ihn temporär zu einem strong Pointer machen, wodurch der Referenzzähler erhöht wird. Benötigt man ihn nicht mehr, kann man ihn wieder zu einem weak Pointer machen.
Dies hat natürlich auch wieder Nachteile. Zum einen muss man genau wissen, wann man welche Pointer benutzen muss und zum anderen muss man immer 2 Fälle beachten. Da der weak Pointer die Referenz nicht erhöht, kann es immer sein, dass der Speicherbereich bereits freigegeben wurde. Man muss also den Fall beachten, dass alles noch da ist und den Fall, dass der Speicher bereits freigegeben wurde und somit eventuell die Daten neu laden muss.

**Hinweis**

ext2 ist nicht von den Zyklen betroffen, da die Inodes nicht auf andere Inodes zeigen können.

## 1.6 Garbage Collection

Während die Referenzzählung eher passiv ist, ist die Garbage Collection aktiv.
Sie überwacht aktiv den Speicher und bereinigt diesen, wenn nötig.
Der Vorteil hierbei ist, dass keinerlei Interaktion vom Anwender nötig ist. Der Nachteil, dass dies natürlich einen großen Aufwand bedeutet und somit Rechenleistung und Speicher benötigt.
Dies kann zum Beispiel mit dem Mark-and-Sweep-Algorithmus erfolgen.
Bei diesem wird von allen noch bekannten und benutzen Objekten ausgehend den Referenzierungen gefolgt. Jedes Referenzierte Objekt wird hierbei markiert. Am Schluss werden alle nicht markierten Objekte freigegeben.
Da hier immer große Bereiche freigegeben werden, fällt die Fragmentierung nicht so stark aus, wie bei der Referenzzählung, wo immer einzelne Objekte gelöscht werden.
Da hier immer noch Fragmentierung entsteht, gibt es noch den Mark-and-Compact-Algorithmus. Dieser geht genau, wie der Mark-and-Sweep-Algorithmus vor, jedoch werden am Ende alle markierten Objekte an eine Stelle im Speicher kopiert. Dadurch ist der freie und belegte Speicher jeweils Zusammenhängend. Dies geschieht natürlich auf Kosten der Geschwindigkeit.
Bei vielen Garbage Collections wird zusätzlich noch ein Generationen Modell verwendet. Hier werden die Objekte in verschiedene Generationen eingeteilt. Jeweils nach ihrer Langlebigkeit. So kann man auf verschiedene Generationen verschiedene Algorithmen anwenden, die am besten passen. Zum Beispiel wäre es für die langlebigen Objekte sinnvoll diese Zusammenhängend zu speichern, somit wäre der Makrk-and-Sweep-Algorithmus geeignet.

# 2 Implementierungen

Im folgenden stelle ich verschiedene Möglichkeiten für die Implementierung der Referenzzählung vor.

## 2.1 “Naive” Implementierung

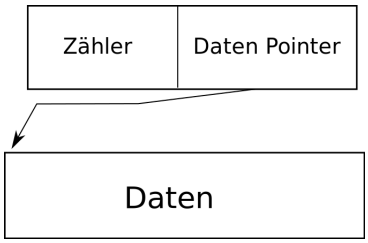
Bei der naiven Implementierung nutzen wir für die Speicherung der Daten eine einfache Struktur. Diese besteht aus einem void-Pointer für die Daten, sowie aus dem Zähler.

```
16 struct memoryObject{
17   void* data;
18   unsigned int refCount;
19 };
```

Listing 2: memoryObject

Dies sieht dann im speicher ungefähr so aus:

Abbildung 1: Speicher




Zusätzlich werden noch Methoden benötigt, um mit dieser Struktur arbeiten zu können.

```
21 struct memoryObject* alloc(size_t);
22 struct memoryObject* getRef(struct memoryObject*);
23 void release(struct memoryObject*);
```

Listing 3: Methoden

Die Methode alloc allokiert hier den Speicher mit der angegeben Größe. Zurückgegeben wird das neue Objekt mit dem Zeiger auf den Speicher, sowie dem Zähler.

```
11 struct memoryObject* alloc(size_t size){
12   struct memoryObject* object = 0;
13
```

```
14    object = (struct memoryObject*) malloc(sizeof(struct memoryObject));
15
16    object->refCount = 1;
17    object->data = malloc(size);
18
19    return object;
20  }
```

Listing 4: Alloc

Zuerst benötigt man die Größe der Struktur als Speicher, welches gleich dem Speicheroverhead ist. Dann setzt man den Referenzzähler auf 1 und holt sich die angeforderte Speichergröße ganz normal mit malloc.

Die nächste Methode ist sehr wichtig. Es handelt sich um die getRef-Methode. Diese muss immer statt einer einfachen Zuweisung benutzt werden, da sonst der Referenzzähler nicht erhöht werden kann.
In der Methode wird einfach der Referenzzähler um 1 erhöht und das Übergebene Objekt zurückgegeben.

```
22  struct memoryObject* getRef(struct memoryObject* object){
23    object->refCount++;
24    return object;
25  }
```

Listing 5: getRef

Die letzte Methode ist release. Dies wird statt dem normalen free genutzt. Dabei heißt ein release nicht auch gleich die Löschung der Daten. Zunächst wird erst der Referenzzähler um 1 dekrementiert. Fällt dieser auf 0, wird das Objekt gelöscht.

```
27  void release(struct memoryObject* object){
28
29    object->refCount--;
30
31    if( object->refCount == 0){
32      free(object->data);
33      free(object);
34
35    }
36  }
```

Listing 6: release

Die Nachteile bei dieser Implementierung sollten klar sein: Vorhandener Code muss extrem angepasst werden. Da es in C nicht möglich ist die Logik des "=" Operators zu verändern, kann man zumindest das getRef nicht vereinfachen. Jedoch muss bei dieser Implementierung immer das Objekt einer Methode übergeben werden, wodurch dessen Signatur geändert werden muss. Dies wäre sehr umständlich und würde zu diversen Fehlern führen. Zusätzlich wäre das Programm sehr an diese eine Implementierung gebunden, wodurch es wiederum aufwändig wäre eine andere zu verwenden.
Die Benutzung würde folgendermaßen aussehen:

```
25 void usage(){
26   struct memoryObject* object = alloc(sizeof(int));
27
28   *((int*)object->data) = 5;
29
30   struct memoryObject* object2 = getRef(object);
31
32   release(object);
33   release(object2);
34 }
```

Listing 7: release

Hier sieht man eine weitere Schwäche dieser Implementierung. Zeile 28 ist sehr unübersichtlich und damit auch sehr anfällig für Fehler, dabei handelt es sich nur um eine einfache Zuweisung.

## 2.2 “Transparente” Implementierung

Ich nenne diese Implementierung die “Transparente” Implementierung, da man hier nur wenig von der Referenzzählung mitbekommt. Man muss nur die neuen Methoden aufrufen, sowie statt der Zuweisung getRef benutzen.
Das Objekt sieht diesmal einfacher aus:

```
15 typedef struct{
16   unsigned int refCount;
17   //void* data;
18 }memoryObject;
```

Listing 8: Objekt

Im Speicher sieht das Ganze dann sehr simpel aus.

Abbildung 2: Speicher


Hier befindet sich nun wieder der Zähler, sowie der Speicher, welchen wir Allokieren. Das besondere hierbei ist, dass wir keinen direkten Pointer auf den Speicherbereich haben, sondern diesen immer errechnen. Dies verringert den Speicheroverhead auf die Zählervariable. Also bei einem Integer in der Regel 4 Byte.
Das Besondere an dieser Implementierung ist, dass man als “Anwender” nie das eigentliche Objekt sieht, sondern immer nur den Pointer auf das Objekt.

Analog zum vorherigen Beispiel haben wir wieder eine alloc, eine getRef und eine release Funktion. Diese unterscheiden sich jedoch durch ihre Parameter und Rückgabewerte.

```
20 void* alloc(size_t);
21 void* getRef(void*);
22 void release(void*);
```

Listing 9: Methoden

Hier sieht man, dass alloc und release die gleiche Signatur, wie malloc und free haben. Dadurch hat man den Vorteil, dass man nur malloc durch alloc und free durch release ersetzen muss ohne die Signaturen anpassen zu müssen.
Das einzige, was noch aufwändig anzupassen ist, sind neue Zuweisungen.

```
11 void* alloc(size_t size){
12
13   size_t overheadSize = sizeof(memoryObject);
14
15   memoryObject* o = 0;
16   char* ptr = 0;
17
18   o = (memoryObject*) malloc(overheadSize + size);
19   ptr = (char*) o;
20   ptr += sizeof(memoryObject);
21
22   o->refCount = 1;
23
24   return (void*)ptr;
25 }
```

Listing 10: alloc

Hier wird zuerst der benötigte Speicher allokiert. Dieser besteht zum einen aus dem Overhead und aus der angeforderten Größe.
Dann benötigen wir einen char-Pointer, da man mit diesem einzelne Bytes zum Pointer addieren bzw. subtrahieren kann.
Dieser Pointer zeigt zunächst auf den Zähler. Addieren wir jedoch die Größe unseres Overheads zu diesem Pointer zeigt dieser nun auf die Daten. Dann wird noch der Zähler auf 1 gesetzt und der Pointer wird zurückgegeben.

```
27 void* getRef(void* pointer){
28
29   memoryObject* o;
30   char* cptr;
31
32   cptr = (char*)pointer;
33   cptr -= sizeof(memoryObject);
34   o = (memoryObject*)cptr;
35
36   o->refCount++;
37
```

```
38    return pointer;
39 }
```

Listing 11: getRef

**Warnung**

Der Pointer muss von der alloc Methode erzeugt worden sein, sonst ist das Verhalten des Programmes undefiniert.

Bei dieser Methode wird nun ein einfacher void Pointer übergeben. Dies sieht dann ungefähr so aus

Abbildung 3: Datenpointer


Zieht man von diesem Pointer nun die Größe des Zählers ab, zeigt der Pointer auf den Anfang des Zählers.

Abbildung 4: Zaehlerpointer

Nun kann dieser ganz einfach erhöht werden.
Die release Methode arbeitet wieder genau so. Der Pointer wird auf den Zähler geschoben, dieser wird verglichen und wenn dieser auf 0 fällt, wird der Speicher wieder freigegeben.

```
41 void release(void* pointer){
42   memoryObject* o;
43   char* cptr;
44
45   cptr = (char*)pointer;
46   cptr -= sizeof(memoryObject);
47   o = (memoryObject*)cptr;
48
49   o->refCount--;
50
51   if( o->refCount == 0){
52     free(o);
53   }
54 }
```

Listing 12: release

## 2.3 C++

Dieses Dokument soll zwar die Referenzzählung für C behandeln, jedoch ist es einen Blick wert, wie weit man die Referenzzählung noch verbessern kann, so dass es beim Anwender kaum noch zu Problemen kommen kann.
Bei C++ hat man den Vorteil, dass man Destruktoren und die Operatorüberladung hat.
Ein Destruktor wird aufgerufen, sobald ein Objekt zerstört wird. Durch die Operatorüberladung kann man die Logik von Operatoren verändern. In diesem Fall ist der Zuweisungsoperator wichtig.
Natürlich hat dies nicht nur Vorteile, was man noch bei der Effizienz sehen wird.
In diesem Beispiel handelt es sich um die Shared Pointer Implementierung von Boost. Diese Bibliothek wurde auch in den neusten C++ Standard aufgenommen und ist deshalb gut geeignet.

```
18 void usage(){
19   shared_ptr<int> ptr1(new int);
20
21   *ptr1= 10;
22
23   shared_ptr<int> ptr2 = ptr1;
24 }
```

Listing 13: Shared Pointer

Hier wird wieder, wie im ersten Beispiel, ein Objekt für die Speicherung benutzt. Dadurch müssen Signaturen wieder angepasst werden. Jedoch können nur so die angesprochenen Vorteile genutzt werden.
In diesem Beispiel sieht man, dass man diesen einfach mit "*" dereferenzieren kann und mit "=" einfach zuweisen kann. Zusätzlich muss man den Speicher nicht mehr manuell freigeben, da das Objekt am Ende der Methode zerstört wird und somit der Destruktor aufgerufen wird, welcher das Aufräumen übernimmt.

# 3 Effizienz

Die Referenzzählung kann eine Effizienzsteigerung bei der Entwicklung bedeuten. Dadurch, dass sich der Entwickler nicht mehr so sehr darum kümmern muss, ob ein Speicherbereich freigegeben werden kann, sondern einfach den Zähler verringert, wenn der Speicher an einer bestimmten Stelle nicht mehr benötigt wird.
Der Nebeneffekt ist, dass die Referenzzählung Rechenleistung benötigt. Dies wird hier am Beispiel der vorgestellten Implementierungen getestet.

**Warnung**

Diese Messwerte sollen weder Repräsentativ sein, noch sind die Implementierungen perfekt. Es soll lediglich eine Tendenz gezeigt werden, wie viel Leistung eine Implementierung benötigen kann.

Der Ablauf ist nun folgendermaßen: Zuerst werden eine bestimmte Anzahl an Integers allokiert. Dann wird dieser Wert einem anderen Pointer zugewiesen und sofort wieder freigegeben. Zum Schluss wird der gesamte Speicher wieder freigegeben.
Dies geschieht jeweils in Schleifen, die 10 000 000 mal durchlaufen werden. Die Testmethoden unterscheiden sich dabei je nach Implementierung immer ein wenig.
Bei der Variante ohne Referenzzählung muss natürlich in der 2. Schleife nichts freigegeben werden, da dies alles manuell geschehen muss.

```
77   for(i = 0; i < ITERATIONS; ++i){
78     pointers[i] = (int*)malloc(sizeof(int));
79   }
```

Listing 14: Alloc

```
82   for(i = 0; i < ITERATIONS; ++i){
83     linkPtr = pointers[i];
84   }
```

Listing 15: Get Reference

```
87   for(i = 0; i < ITERATIONS; ++i){
88     free(pointers[i]);
89   }
```

Listing 16: Release

Die Naive, sowie die Transparente Implementierung, unterscheidet sich in diesem Beispiel nicht.

```
46  for(i = 0; i < ITERATIONS; ++i){
47    pointers[i] = (int*)alloc(sizeof(int));
48  }
```

Listing 17: Alloc

```
51  for(i = 0; i < ITERATIONS; ++i){
52    linkPtr = (int*)getRef(pointers[i]);
53    release(linkPtr);
54  }
```

Listing 18: Get Reference

```
57  for(i = 0; i < ITERATIONS; ++i){
58    release(pointers[i]);
59  }
```

Listing 19: Release

Bei der Implementierung in C++ gibt es eine Besonderheit. Hier muss kein expliziter Aufruf stattfinden, um den Speicher wieder freizugeben. Aus diesem Grund findet die Messung der Zeit statt, nachdem die komplette Methode zu Ende ist und somit die Destruktoren aufgerufen wurden.

```
33  for(i = 0; i < ITERATIONS; ++i){
34    pointers[i].reset(new int);
35  }
```

Listing 20: Alloc

```
38  for(i = 0; i < ITERATIONS; ++i){
39    linkPtr = pointers[i];
40  }
```

Listing 21: Get Reference

Die Ergebnisse wurden auf einem PC mit einem Q6600 mit 2.4GHz 4GB RAM auf einem Ubuntu 12.10 mit einem 3.5.0-24-generic 64Bit Kernel und der gcc Version 4.7.2 erstellt.

Hier sind die Ergebnisse, ohne Optimierungen (-O0)

| | Ohne | Naiv | Transparent | C++ |
|---|---|---|---|---|
| alloc | 0.583308s (0.71) | 1.139242s (0.6) | 0.648253s (0.48) | 1.844750s (0.35) |
| getRef | 032474s (0.04) | 0.296284s (0.16) | 0.313264s (0.23) | 2.017531s (0.39) |
| release | 0.211202s (0.26) | 0.468051s (0.25) | 0.381382s (0.28) | 1.356881s (0.26) |
| Gesamt | 0.826984s | 1.903577s | 1.342898s | 5.219162s |

Und hier mit vielen Optimierungen (-O3)

| | Ohne | Naiv | Transparent | C++ |
|---|---|---|---|---|
| alloc | 0.579606s (0.7) | 1.079139s (0.63) | 0.574550s (0.66) | 1.091072s (0.38) |
| getRef | 0.032213s (0.04) | 0.271743s (0.16) | 0.147616s (0.16) | 1.075680s (0.38) |
| release | 0.210672s (0.26) | 0.370024s (0.22) | 0.196138s (0.21) | 0.695431s (0.24) |
| Gesamt | 0.822490s | 1.720905s | 0.918305s | 2.862182s |

**Hinweis**

Die Zahlen in den Klammern sind der Anteil der benötigten Zeit. Dabei handelt es sich um gerundete Werte, sodass die Summe nicht immer gleich 1 ist.

Bei den Ergebnissen fällt auf, dass die Transparente Implementierung nur minimal länger benötigt, als die Variante ohne Referenzzählung. Außerdem scheint die Optimierung keinen Effekt bei der normalen Variante zu zeigen. Zudem sieht man, dass die naive Implementierung gar nicht so schlecht ist. Dies zeigt, dass die Referenzzählung recht einfach umzusetzen ist, ohne dabei gleich ein langsames Programm zu haben.
Wie zu erwarten war, ist die C++ Implementierung am langsamsten. Dafür ist diese jedoch die sicherste und umfangreichste Variante.

Zusammenfassend kann man also sagen, dass die Referenzzählung ein gutes und einigermaßen simples Werkzeug ist, gemeinsame Speicherbereiche zu verwalten und ist manchmal sogar unumgänglich. Dabei ist diese nicht perfekt und bedarf noch zusätzliche Arbeit durch den Entwickler. Zudem gibt es mehrere mögliche Implementationen, die alle ihre eigenen Vor- und Nachteile haben.

# 4 Quellen


- http://e2fsprogs.sourceforge.net/ext2intro.html
- http://www.eosgarden.com/en/articles/c-refcount/
- Boost/STL/Java/Python Dokumentation
- http://en.wikipedia.org/wiki/Reference_counting