Garbage Collection

Alexander van Renen

Fakultät für Informatik
Technische Universität München
Boltzmannstr. 3
85748 Garching, Deutschland
renen@in.tum.de

Abstract: Die meisten modernen Programmiersprachen, wie auch C#, verwenden einen Garbage Collector. In dieser Ausarbeitung soll die Funktionsweise eines solchen Systems, sowie dessen effektiver Einsatz erläutert werden.

Inhaltsverzeichnis

1 Einführung

Vor beinahe 60 Jahren entwickelte John von Neumann eine Architektur für den Bau einer Rechenmaschine. Dieses Modell – heute bekannt als die “von-Neumann-Architektur“ – ist in beinahe allen heutigen Rechnern noch enthalten. Selbstverständlich gab es über die Zeit hinweg einige Verfeinerungen, doch der grundlegende Aufbau ist gleich geblieben.

Ein Teil dieses Modells war die Speichereinheit, welche die Programme, aber auch die eigentlichen Daten für die Maschine aufbewahrt. Da die beiden primären Optimierungsziele für den Speicher, nämlich dessen Größe und Zugriffsgeschwindigkeit, sich gegenseitig ausschließen, wurde eine hierarchische Speicherarchitektur eingeführt. Es gibt schnelle Speicher, wie den Cache, der aber aufgrund seiner Platzierung auf der CPU nur sehr wenig Speicherplatz bietet. Auf der anderen Seite gibt es große Speicher, wie Festplatten, die dafür sehr lange Zugriffszeiten haben.

Dazwischen liegt der Arbeitsspeicher, welcher in jedem heutigen Computer eingebaut ist. In diesem Teil der Speicherhierarchie werden beim Starten eines Programms dessen Code und Daten, auf welchen während dem Ausführen gearbeitet wird, kopiert. In der folgenden Ausarbeitung wird gezeigt werden, wie ein Programm diesen Speicher verwaltet und wie man diese Verwaltung durch die Einführung eines Garbage Collectors vereinfachen kann.

1.1 Speicherverwaltung

Vereinfacht gesagt wird in heutigen Programmen der zur Verfügung stehende Speicherbereich in vier Teile aufgeteilt (vgl. Abb. 1).

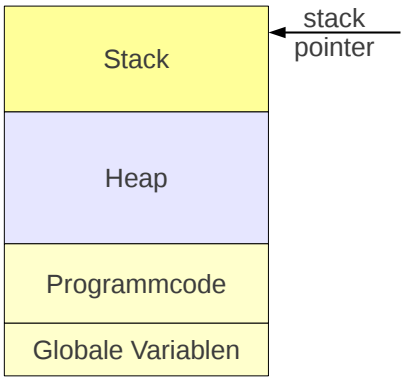

Abbildung 1: Speicheraufbau

- Programmcode: In diesen Bereich wird beim Start des Programms der Programmcode, welcher zu dessen Ausführung benötigt wird, kopiert. Dieser Bereich verändert sich über die Laufzeit des Programms in der Regel nicht.
- Globale Variablen: Statische oder globale Variablen sind über die komplette Laufzeit verfügbar. In C# sind das vor allem Variablen, die als $static$ deklariert werden. Sie bekommen bei Programmstart einen festen Platz im Bereich der globalen Variablen und behalten diesen über die komplette Laufzeit.
- Stack: Der Stack wird aus so genannten Frames aufgebaut. Bei jedem Funktionsaufruf wird ein neuer Frame direkt hinter dem jeweils aktuellen Frame angelegt. Dazu wird der Stackpointer, der immer auf das Ende des aktuellen Frames zeigt, an das Ende des neuen Frames verschoben. Ein Frame beinhaltet alle lokalen Variablen einer Funktion sowie deren Parameter. Sobald ein Funktionsaufruf durchgeführt wurde (also ein $return$ Statement ausgeführt wird), wird der Stackpointer wieder auf das Ende des vorherigen Frames gesetzt.
- Heap: Der Heap ist der für diese Arbeit interessanteste Bereich, da hier der Speicher nicht automatisch durch das Verschieben des Stackpointers verwaltet wird,

sondern vom Programmierer durch spezielle Funktionen explizit angefordert und wieder zurückgegeben werden muss. Dieser Speicher wird hauptsächlich für große Daten, Daten mit variabler Länge oder Daten, die an mehren Orten in einem Programm benötigt werden, verwendet. Sollte Speicher angefordert, aber nicht mehr zurückgegeben werden, entstehen Probleme, da der noch verfügbare Speicher immer geringer wird.

Listing 1: Code Beispiel

```
void foo()
{
   void* ptr = malloc(8128);

   /*benutze ptr*/

   return;
}
```

'Listing 1' zeigt eine kleine Beispiel Funktion in der Sprache C. Beim Aufrufen dieser Funktion wird zuerst der Stackpointer so verschoben, dass genug Platz für alle lokalen Variablen entsteht. Im gegebenen Beispiel ist dies die lokale Variabale ptr. Jene ist vom Typ $void*$, welcher im Grunde nur eine beliebige Speicheradresse enthält. Die Funktion $malloc$ wird in C dafür benutzt, einen Speicherbereich auf dem Heap anzufordern. Das Argument, welches ihr übergeben wird (im vorliegenden Fall 8128), stellt die Größe des angeforderten Speicherbereichs dar. Es wird also ein Speicherbereich der Größe 8128 angefordert und dessen Adresse in ptr gespeichert. Nachdem die Funktion mit dem Speicher gearbeitet hat, wird das $return$ Statement ausgeführt, welches den Stackpointer wieder auf das Ende des Frames der aufrufenden Funktion setzt.

Durch das Umsetzten des Stackpointers wird der Speicherplatz, der für ptr gebraucht wurde (um die Adresse selbst zu speichern), frei und steht für den nächsten Funktionsaufruf wieder zur Verfügung. Der Speicherbereich auf dem Heap, der durch $malloc$ angefordert wurde, geht allerdings verloren, da er durch die Funktion nicht freigegebn wurde. Dazu müsste der Programmierer noch die Funktion $free$ aufrufen. Diese bekommt als Argument die Adresse des freizugebenden Speicherbereichs und sorgt dafür, dass der ihr übergebene Bereich im weiteren Verlauf wieder benutzbar ist. Ohne den Aufruf von $free$ entseht ein Memoryleak, da bei jedem Aufruf von foo effektiv 8128 Byte an Speicher verloren gehen.

Dieser Umstand ist bei lange laufenden Programmen aus offensichtlichen Gründen nicht vorteilhaft, weswegen es unbedingt notwendig ist, jeden Speicherbereich, der durch $malloc$ alloziert wurde, wieder durch $free$ freizugeben. Dies ist allerdings nicht immer so einfach wie in der Beispielfunktion, die im wesentlichen aus drei Zeilen Code besteht. Man denke hier an eine komplizierte Funktion mit mehreren alternativen Kontrollflüssen. Vor jedem $return$ muss der Programmierer daran denken $free$, für alle angeforderten Speicherbereiche aufzurufen. Einem Wartungs-Programmierer, der Jahre später eine Funktion leicht abändern muss, aber nicht die Zeit hat, die komplette Logik in der Funktion zu verstehen, könnte leicht ein Fehler unterlaufen. Ein weiteres Problem stellen Ausnahmen (Exceptions) dar. Diese können sogar ein Memoryleak verursachen, wenn sie in einer Funktion,

die von der gerade betrachteten Funktion aufgerufen wurde, geworfen werden. Der Grund hierfür ist, dass durch eine geworfene Ausnahme eventuell ein Aufruf von $free$ übersprungen wird. Die druch Ausnahmen verursachten Memoryleaks, versucht man durch sogenanntes ”Exception Handling“ zu beseitigen ([Tut]). Neben Ausnahmen ist sind Fälle zu beachten, in denen der angeforderte Speicher nicht nur in einer einzelnen Funktion verwendet wird, sondern in vielen Teilen des Programms. Dadurch wird das Richtige manuelle freigeben des Speichers weiter erschwert. Zuletzt sollte man auch noch an Programme mit mehreren Threads denken, in welchen es Speicherbereiche gibt die zwischen den Threads geteilt und somit gemeinsam verwaltet werden. Auch hier wird schnell klar, dass es einiges an Aufwand erfordert, den Speicher immer an der richtigen Stelle wieder freizugeben.

Wie man sieht, entstehen durch die manuelle Verwaltung des Heaps einige Schwierigkeiten. Ein Weg, diesen zu begegnen, ist die Einführung eines Garbage Collectors. Alternativen dazu findet man beispielsweise in C++, wo versucht wird, das Problem durch “Resource-Managing Classes” zu lösen ([Mey10]).

1.2 Prinzip des GC

Wie man im vorigen Abschnitt deutlich sehen konnte, gibt es einige Probleme mit der manuellen Speicherverwaltung. Die Idee einer Lösung für das Problem des Speicherverlustes ist sehr einfach. Man führt ein System ein – den Garbage Collector. Dieser soll selbständig Speicherbereiche, welche nicht mehr benötigt werden, finden und wieder freigeben. Mit einem Garbage Collector kann das Problem der manuellen Speicherverwaltung beseitigt und somit die Arbeit des Programmierers erleichtert werden.

In den meisten älteren Programmiersprachen – wie A, B oder C – gibt es keinen Garbage Collector und man muss sich, wie es am Beispiel von C ersichtlich ist, selbst darum kümmern. Der Grund dafür liegt hauptsächlich darin, dass ein Garbage Collector, wie man später noch sehen wird, einiges an Ressourcen verbraucht. Da es aber heutzutage bei vielen Systemen immer unwichtiger wird, seine Programme auf Kosten von Design, Portability oder eben Speicherverwaltung effizienter zu machen, beinhalten die meisten modernen Programmiersprachen einen Garbage Collector. So auch die Programmiersprache C# oder vielmehr der Managed C# Code, der von der CLR (Common Language Runtime) von .NET verwaltet wird, welche einen Garbage Collector benutzt.

2 Funktionsweise

In diesem Kapitel soll nun, nachdem der Grund für die Einführung eines Garbage Collectors und dessen grobe Aufgabe erläutert wurde, beschrieben werden, wie ein solches System realisiert werden kann. Am Ende soll noch ein kleiner Ausblick auf mögliche Verbesserungen gegeben.

2.1 Idee

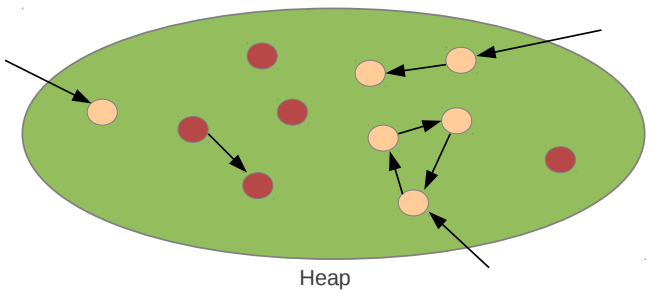

Abbildung 2: Abstrakte Darstellung des Heaps

In Abbildung 2 ist eine abstrakte Darstellung des Heaps zu sehen. An dieser soll nun die Funktionsweise von Garbage Collection veranschaulicht werden. Das große Oval stellt eine Momentaufnahme des gesamten Heaps – inklusive des in grün dargestellen freien Speichers – des Programms zur Laufzeit dar. Die kleineren Kreise innerhalb des Heaps stellen Objekte (zusammengehörige Speicherbereiche) dar. Jedes von ihnen wurde zu einem früheren Zeitpunkt von einem Programmierer erstellt. Jeder Pfeil stellt eine Referenz von einem Objekt auf ein anderes dar. Durch die Farben ist gekennzeichnet, welche Objekte noch von dem Programm benötigt werden und welche gelöscht werden können. Die rot eingefärbten Objekte sind offensichtlich nicht mehr von Nutzen für das Programm. Dies sollte intuitiv klar sein, denn sobald ein Objekt nicht mehr durch eine Kette von Referenzen erreichbar werden kann, ist es auch nicht mehr möglich auf dieses zuzugreifen.

Mit den folgenden zwei Definitionen soll dieser Sachverhalt noch verdeutlicht werden. Zuerst soll der Begriff Stamm eingeführt werden.

Definition 1 (Stamm) *Eine Referenz ist ein Stamm genau dann wenn*

- *sie eine lokale Variable auf dem Stack ist oder*
- *sie eine statische Variable ist.*

Man sollte beachten, dass damit Member Variablen in einer $struct$ automatisch auch als Stämme zählen, da diese in C# auf dem Stack liegen. In Abbildung 2 sind die drei Pfeile, die von außerhalb des Heaps nach innen zeigen, Stämme.

Mit der Notation des Stammes kann man nun festlegen, was es für ein Objekt heißt, lebendig zu sein.

Definition 2 (lebendig) *Lebendige Objekte werden von einem Programm noch benötigt. Ein Objekt ist lebendig genau dann wenn*

- *ein Stamm auf es zeigt.*
- *ein lebendiges Objekt auf es zeigt.*

Durch diese rekursive Definition des Begriffs “lebendig“ ist nun klar festgelegt, welche Objekte ein Programm noch braucht und welche dagegen nicht. Die Aufgabe des Garbage Collectors kann also wie folgt formuliert werden: Finde und zerstöre alle Objekte, die nicht lebendig sind.

2.2 “Mark and Sweep“-Algorithmus

Im folgenden Abschnitt soll anhand einer vereinfachten Übersicht die Funktionsweise des Garbage Collectors, der in .NET verwendet wird, erklärt werden. Der Kern des Garbage Collectors ist der ”Mark and Sweep“-Algorithmus, welcher auch in anderen Laufzeitumgebungen, wie zum Beispiel in der von Java, verwendet wird. Der ”Mark and Sweep“-Algorithmus soll anhand von Abbildung 3, einer etwas abgewandelten Darstellung des Heaps, erklärt werden.

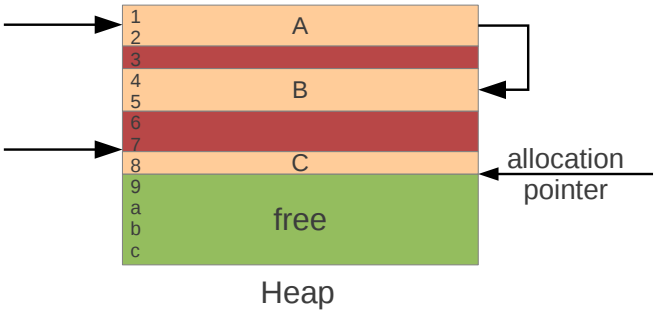

Abbildung 3: Darstellung des Heaps

Dieser Beispiel-Heap hat aus Gründen der Übersichtlichkeit nur eine Größe von 12 Feldern (Bytes). Wie auch in der vorherigen Abbildung stellen die Pfeile Referenzen dar. Hier sind die Pfeile auf der linken Seite Stämme, da sie nicht aus dem Heap, sondern aus dem Stack oder dem Bereich mit den globalen Variablen kommen. Es zeigt also ein Stamm auf Objekt A und Objekt C, weswegen diese lebendig sind und somit nicht gelöscht werden dürfen. Objekt B ist ebenfalls lebendig, da das lebendige Objekt A eine Referenz auf dieses hält. Die beiden rot markierten Speicherbereiche (an den Adressen 3 und 6-7) werden nicht mehr benötigt, da sie nicht lebendig sind.

Der neu eingeführte $Allocation\ Pointer$ dient zur Erstellung neuer Objekte. Wenn in C# ein neues Objekt (oder ein anderer Referenztyp) erstellt werden soll, dann wird zunächst die benötigte Größe für das neue Objekt in Byte bestimmt. Der $Allocation\ Pointer$ wird dann um diesen Wert (im Bild) nach unten verschoben, um genügend Platz für das neue Objekt zu machen. Jenes wird dann in den Bereich zwischen der alten und neuen Position des $Allocation\ Pointer$ gelegt. Wie man sieht, läuft das Anfordern von Speicher auf dem Heap lediglich auf eine Verschiebung des $Allocation\ Pointer$ hinaus. Dies ist eine sehr

schnelle Methode, um neuen Speicher auf dem Heap anzufordern. In anderen Sprachen – wie Beispielsweise C – wird eine Liste mit allen freien Speicherblöcken, welche bei einer Speicheranfrage durchsucht werden muss, verwaltet.

Mit der Definition von lebendigen Objekten kann nun der "Mark and Sweep"-Algorithmus definiert werden.

Algorithmus 1 *Der "Mark and Sweep"-Algorithmus besteht aus vier Teilen:*

1. *Pausiere die Ausführung des Programms.*
2. *Markiere alle lebendigen Objekte.*
3. *Gib den Speicher aller toten Objekten frei.*
4. *Nimm die Ausführung des Programms wieder auf.*

Im Folgenden soll dieser Algorithmus erläutert werden.

- Schritt 1

  Zunächst muss das eigentliche Programm, welches der Programmierer erstellt hat, pausiert werden. Dies ist notwendig, da es unabdingbar ist, dass der Speicher während der nächsten Schritte nicht vom Benutzercode verändert wird. Was hierbei ins Auge sticht, ist die Notwendigkeit eines möglichst schnellen Algorithmus, damit die Ausführung des Programms nicht zu lange gestoppt wird.

- Schritt 2

  In Schritt 2 werden alle lebendigen Objekte markiert. Dieser Schritt ist leicht nachvollziehbar sein, wenn man sich die Definition von lebendig ins Gedächtnis ruft. Der Garbage Collector benötigt zunächst eine Liste mit allen Stämmen. Stämme liegen entweder im Speicherbereich für globale Variablen oder auf dem Stack. Der Garbage Collector muss also einmal zum Programmstart jede Referenz in den globalen Variablen finden und in seine Liste von Stämmen aufnehmen. Die Referenzen auf dem Stack werden gefunden, indem die Metadaten von allen Frames vor dem $Stack\ Pointer$ analysiert werden. Nun kann die Definition von lebendig angewendet werden, indem man – beginnend von den Stämmen – alle Objekte besucht und dabei allen Referenzen folgt. Wenn man sich die Objekte als Knoten und die Referenzen als Kanten in einem Graph vorstellt, kann man das Problem durch Tiefen- oder Breitensuche lösen. Jedes gefundene Objekt ist lebendig. Dies wird durch das Setzen eines Flags markiert. Dieser Schritt wird als "Mark"-Phase des Algorithmus bezeichnet. Nach ihr sind alle noch lebendigen Objekte markiert.

- Schritt 3

  Nun sollen alle Objekte, die nicht markiert sind, ergo nicht mehr benötigt werden, gelöschen werden. Dazu wird der komplette Speicher wie er in Abbildung 3 dargestellt ist, von oben (also Adresse 1) nach unten bis zum $Allocation\ Pointer$ (also Adresse 8) durchsucht. Da der $Allocation\ Pointer$ immer (in der Abbildung) unter allen erstellten Objekten ist, ist es ausreichend, wenn die Suche hier abgebrochen wird. Das Löschen geschieht nun implizit dadurch, dass alle lebendigen Objekte auf dem Heap nach oben geschoben werden und so die Lücken aus

nicht mehr benötigten Objekten aufgefüllt werden. Nachdem alle lebendigen Objekte zusammengeschoben wurden, wird der *Allocation Pointer* an das neue Ende der lebendigen Objekte gesetzt. Dieser Schritt stellt die ”Sweep“-Phase des Algorithmus dar.

- Schritt 4

  Als letztes kann die Ausführung des eigentlichen Programms wieder fortgeseetzt werden.

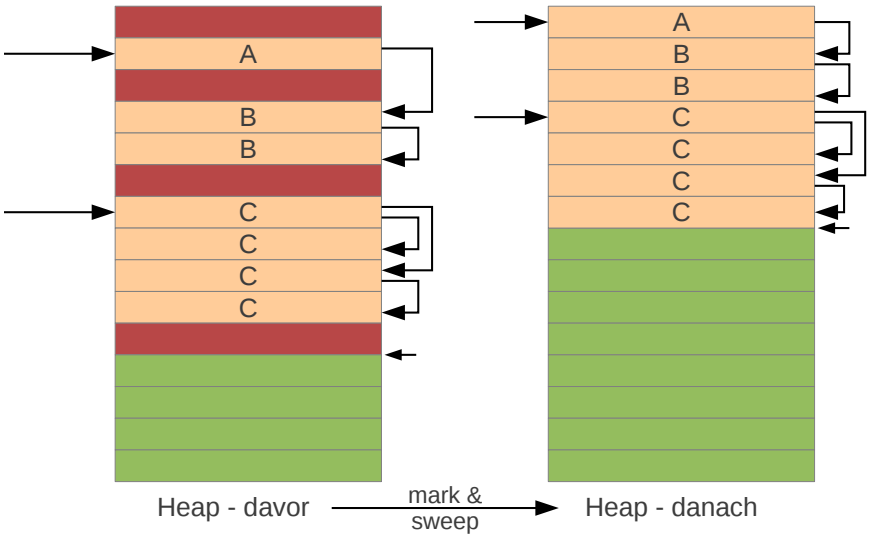

Abbildung 4: Beispiel für die Sweep Phase

Dieses Vorgehen, speziell Schritt 3, soll nun an einem Beispiel klarer gemacht werden (vergleiche hierzu Abbildung 4). Diese zeigt auf der linken Seite den Zustand des Heaps nach der ”Mark“-Phase und auf der rechten Seite den Zustand des Heaps nach der ”Sweep“-Phase. Wie man auf einen Blick erkennen kann, ist der Anteil an freiem Speicher gestiegen. Auf beiden Seiten sind jeweils wieder die lebendigen Objekte in orange, die nicht mehr benötigten Objekte in rot und der freie Speicher in grün gehalten. Auch hier stellen die Pfeile wieder Referenzen dar. Der kleine Pfeil der auf beiden Seiten von rechts kommend den belegten Speicher vom freien Speicher trennt ist der *Allocation Pointer*. Wie man sieht, wird der Platz, den die lebendigen Objekte einnehmen, durch eine Verschiebung nach oben komprimiert und dadurch auch die nicht mehr benötigten Objekte gelöscht.

2.3 Bewertung und Verbesserung

Im letzten Abschnitt wurde gezeigt, wie der ”Mark and Sweep“-Algorithmus den Speicher von nicht mehr benötigten Objekten befreien kann. Um den Speicher dauerhaft sauber zu halten, muss der ”Mark and Sweep“-Algorithmus immer wieder ausgeführt werden. Das Problem hierbei ist, dass die Ausführung zeitlich sehr lange dauert ist. Sowohl Schritt 2, in dem alle Referenzen verfolgt werden müssen, als auch Schritt 3, in dem große Speicherteile kopiert werden müssen, kosten viel Zeit. In dieser Zeit kann das eigentlich Programm keine Fortschritte machen, da es pausiert ist, weswegen man die Ausführung des ”Mark and Sweep“-Algorithmus so lange wie möglich verzögert. Das bedeutet meistens, dass der Algorithmus erst ausgeführt wird, sobald eine Speicheranforderung nicht mehr erfüllt werden kann. Man könnte aber auch ein Konzept benutzen, das in Phasen aktiviert wird, in denen das Programm auf Benutzereingaben, Netzwerkpakete oder ähnliches wartet.

Da man sich allerdings nicht auf Sonderfälle wie diese stützen kann, wurde dieser Algorithmus nicht in der Rohform, die bis jetzt beschrieben wurde, in .NET implementiert, sondern eine weiterentwickelte Form verwendet. Diese Weiterentwicklung teilt den Heap in drei Generationen auf. Jede Generation enthält ungefähr gleich alte Objekte. In Abbildung 5 ist gezeigt, wie diese Generationen entstehen. Neue Objekte werden immer in Generation 0 eingefügt. Der ”Mark and Sweep“-Algorithmus filtert die nicht mehr benötigten Objekte aus einer Generation und erhöht deren Generationsnummer. In .NET werden 3 Generationen verwendet (Java dagegen verwendet nur 2), folglich kann die Generationsnummer von Objekten aus Generation 2 nicht mehr weiter erhöht werden.

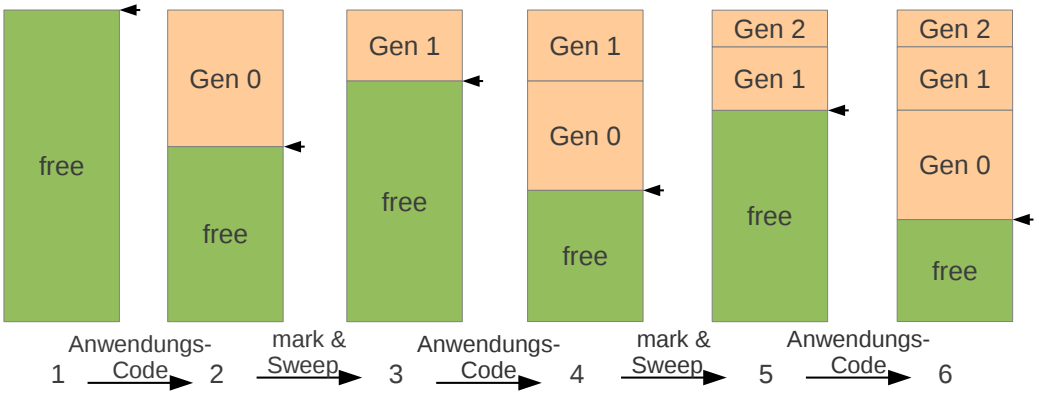

Abbildung 5: Generationen Prinzip

Durch Beobachtungen hat man festgestellt, dass viele Objekte nur sehr kurzlebig sind ([MSDa]). Auf der anderen Seite werden Objekte, die es schon sehr lange gibt, nur noch selten gelöscht. Die Idee ist nun, den ”Mark and Sweep“-Algorithmus nicht mehr auf den gesamten Heap anzuwenden, sondern nur auf eine der Generationen. Beim Anwen-

den auf Generation 0 kann man sich sehr viel Arbeit sparen, indem man nur Stämme und Referenzen innerhalb von Generation 0 verfolgt. Dies ist fast ausreichend, um alle lebendigen Objekte in Generation 0 zu finden, da Objekte aus Generation 1 und 2 nur auf Objekte aus Generation 0 zeigen können, wenn sie sich seit dem letzten "Mark and Sweep"-Algorithmus Aufruf verändert haben, weil sie schließlich älter sind. Oder in anderen Worten: Da es die Objekte aus Generation 0 zum Zeitpunkt der letzten Ausführung des "Mark and Sweep"-Algorithmus noch nicht gab, müssen sich die Objekte aus Generation 0 und 1 seid diesem Zeitpunkt verändert haben, um überhaupt auf sie zeigen zu können. Es werden also zusätzlich noch jene Objekte aus Generation 1 und 2 verfolgt, die sich seit dem letzten Aufruf von Mark and Sweep verändert haben. So können alle lebendigen Objekte in Generation 0 gefunden werden.

Durch diesen Trick kann das Anwenden des "Mark and Sweep"-Algorithmus auf Generation 0, welche am öftesten überprüft werden muss, zeitlich viel schneller realisiert werden. Beim Anwenden des "Mark and Sweep"-Algorithmus auf Generation 2 gibt es keine älteren Objekte, die nur bei Veränderung betrachtet werden müssen, weswegen man sich hier nichts sparen kann. Allerdings werden Objekte aus dieser Generation nur selten gelöscht, weswegen man den Algorithmus nicht oft auf diese Generation anwenden muss. Bei Generation 1, welche die moderat alten Objekte beinhaltet, kann man eine Mischform anwenden, bei der man nur die veränderten Objekte aus Generation 2 anschaut, aber alle aus Generation 0 und 1.

Durch die Nutzung von Generationen kann die Laufzeit also wesentlich verbessert werden.

3 Anwendung des GC in C#

Nachdem nun die Funktionsweise erklärt ist, wird im Folgenden auf die Auswirkungen des Garbage Collectors auf den C# Programmierer eingegangen. Es wird erklärt wie mit der $dispose$ Methode nicht verwaltete Ressourcen freigeben werden können. Ausßerdem wird eine kleine Sammlung mit Hinweisen zum effizienten Umgang mit dem Garbage Collector bereitgestellt.

3.1 Das $IDisposeable$ Interface

Der Garbage Collector verwaltet nur den Speicher für ein Programm, andere Ressourcen wie Fensterhandles, Dateihandles oder Streams werden nicht vom Garbage Collector verwaltet. Diese müssen explizit freigegeben werden. Ausserdem ist es bei kritischen Ressourcen – wie Dateihandles – meist wünschenswert sie nicht erst bei der nächsten Aussfürung des Garbage Collectors freizugeben, sondern sofort. Dazu kann das $IDisposable$ Interface benutzt werden, in welchem eine $dispose$ Methode definiert ist. Die Ressourcen werden dann freigeben, indem die $dispose$ Methode vom Benutzer explizit aufgerufen wird. Bei der Implementierung dieser Methode sollten folgende Dinge beachtet werden:

- Wenn die Klasse, für die die $dispose$ Methode implementiert wird, andere Klassen als Member hat, dann muss sie auch deren $dispose$ Methode aufrufen.
- In der $dispose$ Methode selbst muss immer auch die $dispose$ Methode der Basis Klasse aufgerufen werden, sofern diese ebenfalls das $IDisposable$ Interface implementiert hat.
- Das mehrfache Aufrufen der $dispose$ Methode sollte nicht zu Ausnahmen oder anderen ungültigen Zuständen führen. Dies ist am leichtesten mit einer einfach boolschen Variable zu erreichen, die sich merkt, ob das Objekt schon "$disposed$" ist.
- Eine Klasse, die eine $dispose$ Methode implementiert, sollte immer auch einen $finalizer$ implementieren. Ein $finalizer$ wird vom Garbage Collector aufgerufen, sobald das Objekt gelöscht wird. Wenn der Benutzer keinen eigenen $finalizer$ implementiert, dann wird ein Standard $finalizer$ aufgerufen, der zum einen viel Zeit kostet und zum anderen nicht die Gewünschten Aufgabe – das freigeben der unverwalteten Ressourcen – erfüllt. In dem vom Benutzer implementierten $finalizer$, wird dann die $dispose$ Methode aufgerufen, welche sich um die artgerechte Freigabe der Ressourcen kümmert.
- Sofern die $dispose$ Methode explizit aufgerufen wurde, bevor das Objekt vom Garbage Collector aufgesammelt wird, ist es unnötig für den Garbage Collector, den $finalizer$ des Objekts aufzurufen. Diesen überflüssigen Aufruf kann man umgehen, indem man die $GC.SuppressFinalize$ Methode aufruft, welche dem Garbage Collector mitteilt, dass der $finalizer$ des Objekts nicht mehr aufgerufen werden muss.

In Listing 2 ist eine möglichst einfache Klasse gezeigt, die das $IDisposable$ Interface implementiert. Dieses Beispiel basiert auf [MSDd]. Für genauere Details über den $finalizer$ sei auf [MSDe] verwiesen.

Listing 2: Beispiel für das $IDisposable$ Interface

```
public class MyResource: IDisposable
{
    private bool disposed = false;

    // wird vom benutzer aufgerufen
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    // param disposing == true  => wird vom Benutzer aufgerufen
    // param disposing == false => wird vom GC aufgerufen
    private void Dispose(bool disposing)
    {
        if(!this.disposed) {
            // gib die Ressourcen frei
        }
        disposed = true;
```

```
        }

        // finalizer wird vom Garbage Collector aufgerufen
        ~MyResource()
        {
            Dispose(false);
        }
    }
```

3.2 Die *using* Anweisung

Das Löschen von Objekten in C# ist nicht deterministisch, da der Garbage Collector aus der Sicht des Benutzer Programms zu unbestimmten Zeiten läuft. Dies hat zur Folge, dass zum Beispiel das Schließen eines Dateihandles bis zum nächsten Aufruf des Garbage Collectors warten muss. Allerings sind Dateihandles kritische Ressourcen, die in der Regel so schnell wie möglich freigegeben werden sollten. Dies wird, wie weiter oben beschrieben, mit der *dispose* Methode erledigt. Um den Umgang damit zu vereinfachen, wurde die *using* Anweisung eingeführt, mit welcher man angeben kann, wann das Objekt nicht mehr benötigt wird.

Listing 3: Beispiel für *using* Anweisung

```
Font font = new Font("Arial", 10.0f);
using (font)
{
    // use font
}
```

Listing 3 zeigt einen Codeausschnitt, der die *using* Anweisung benutzt. Das Objekt kann nur innerhalb des *using* Blocks verwendet werden. Wenn eine Ausnahme innerhalb des Blocks geworfen wird oder dessen Ende erreicht wird, wird das Objekt gelöscht. Da zum freigeben der Ressourcen bekannterweise die *dispose* Methode benötigt wird, muss die Klasse des Objekts das *IDisposable* Interface implementiert haben.

3.3 Hinweise zum Umgang mit dem GC

Im Folgenden ist eine kleine Sammlung von Hinweisen zusammengestellt, die man beachten sollte, um den Garbage Collector in C# optimal auszunutzen.

- **Sehr viele Zuordnungen**
  In C# wird, wie weiter oben beschrieben, der Speicher für neue Objekte bereitgestellt, indem lediglich der *Allocation Pointer* verschoben wird. Somit ist das Erstellen von neuen Objekten in C# sehr effizient. Allerdings sollte man zwei Dinge im Auge behalten:
  - Zum Ersten ist das Erstellen zwar schnell, aber diese Kosten werden wieder zurückgefordert, wenn der Speicher durch den “Mark and Sweep ”-Algorithmus

aufgeräumt wird. Und je mehr Objekte angelegt werden, desto früher ist eine solche Aufräumaktion notwendig.
  - Zum Zweiten sollte man immer beachten, wie Funktionen ihre Aufgabe erfüllen. Es kann oft sein, dass eine einzige Codezeile eine Vielzahl von neuen temporären Objekten erstellt, die eigentlich nicht von dem Benutzer selbst benötigt werden. Solche Funktionsaufrufe verkürzen auch die Zeit, bis der Garbage Collector wieder aktiv werden muss.

- **Zu große Zuordnungen**
  In vielen anderen Sprachen ist das Anfordern von neuem Speicher vom Heap nicht so schnell wie in C#. Deswegen wird oftmals ein so genannter Memory-Pool eingesetzt. Dazu wird ein großer Speicherblock auf dem Heap angefordert und dieser dann manuell verwaltet. Dies kann viele teure Speicheranfragen sparen und damit die Performance stark verbessern. In C# ist so etwas nicht nötig, da Speicher anfordern schnell ist. Oft ist es sogar hinderlich, denn dadurch wird viel Speicher unnötig verbraucht. Diese macht den Spielraum des Garbage Collectors kleiner, wodurch die Ausführung des "Mark and Sweep"-Algorithmus früher notwendig wird.

- **Datenstrukturen mit vielen Referenzen**
  Eine zusätzliche Referenz bedeutet eine zusätzliche Kante in dem Graphen, der bei jedem "Mark and Sweep"-Algorithmus Aufruf, abgelaufen werden muss. Dies betrifft auch die Stämme. Man sollte versuchen, Stämme, die nicht mehr benötigt werden, möglichst schnell wieder loszuwerden. Hier hilft auch die $using$ Anweisung.

- **Schlechtes Verhalten des Cache**
  Man denke sich ein Programm, in welchem es eine Listenartige Datenstruktur gibt. Diese wird zu Beginn erstellt und nach und nach mit Daten aufgefüllt. Wenn alle notwendigen Daten gesammelt sind, wird eine komplizierte Berechnung mit der Liste durchgeführt. Das Problem hier ist, dass die Objekte auf dem Heap in C# nach ihrem Alter geordnet sind. Also werden die Listenelemente sehr stark im Speicher verteilt sein, was zu einem schlechten Cache verhalten führen kann.

- **Viele moderat langlebige Objekte**
  Ein oft unvermeidbares, aber schwerwiegendes Problem stellen Objekte dar, deren Lebenszeit so lange ist, dass sie nicht mehr in Generation 0 sind, aber sehr bald nachdem sie in Generation 1 verschoben wurden, gelöscht werden. Das Problem ist, dass die Anwendung des "Mark and Sweep"-Algorithmus auf Generation 1 wesentlich langsamer ist als auf Generation 0. Analog gilt dies natürlich auch für Objekte, die gerade so in Generation 2 rutschen. Objekte dieser Gruppe sollte man versuchen zu vermeiden.

Literatur

[KÏ0] Andreas Kühnel. *Visual C# 2010.* Galileo Computing, 2010.

[Mey10] Scott Meyers. *Effective C++*. 2010.

[MSDa] Microsoft Developer Network MSDN. *Garbage Collector-Grundlagen und Tipps zur Leistung*. http://msdn.microsoft.com/de-de/library/ms973837.aspx.

[MSDb] Microsoft Developer Network MSDN. *IDisposable-Schnittstelle*. link: http://msdn.microsoft.com/de-de/library/system.idisposable(v=vs.80).aspx.

[MSDc] Microsoft Developer Network MSDN. *IDisposable.Dispose-Methode*. http://msdn.microsoft.com/de-de/library/system.idisposable.dispose(v=vs.80).aspx.

[MSDd] Microsoft Developer Network MSDN. *Implementieren einer Dispose-Methode*. link: http://msdn.microsoft.com/de-de/library/fs2xkftw(v=vs.80).aspx.

[MSDe] Microsoft Developer Network MSDN. *Object.Finalize-Methode*. http://msdn.microsoft.com/de-de/library/system.object.finalize(v=vs.80).aspx.

[MSDf] Microsoft Developer Network MSDN. *using-Anweisung (C#-Referenz)*. link: http://msdn.microsoft.com/de-de/library/yh598w02(v=vs.80).aspx.

[Tut] C++ Language Tutorial. *Exception handling*. http://www.cplusplus.com/doc/tutorial/exceptions/.