Garbage-Collection in C++

Daniel Bausch

Technische Universität Darmstadt

3. August 2005

Blockseminar: Fortgeschrittene Konzepte in C++ (SS05)

Inhaltsverzeichnis

1 Einleitung

Ein immer wieder heiß diskutiertes Thema in Bezug auf die zukünftige Entwicklung der Programmiersprache C++ ist die mögliche Einführung einer sprachgestützten automatischen Speicherverwaltung, auch als Garbage-Collection (Müllabfuhr) bezeichnet.

Die herkömmliche Speicherverwaltung mit **new** und **delete** ist zwar vom Konzept her sehr einfach zu verstehen, aber in der Praxis fehleranfällig und mit erheblichem Zusatzaufwand bei der Softwareentwicklung verbunden. Garbage-Collection stellt eine Alternative dar, die die problematische Freigabeoperation von einem völlig autonomen, teilweise im Hintergrund ablaufenden, Prozess erledigen lässt.

Eine Frage dabei ist, wie die Garbage-Collection feststellen kann, welche Objekte nicht mehr benötigt werden. Eine automatische Speicherverwaltung, wenn man sie schon mal hat, bietet aber zusätzlich noch andere interessante Möglichkeiten.

2 Herkömmliche vs. Automatische Speicherverwaltung

2.1 Probleme der herkömmlichen Speicherverwaltung

Mit der herkömmlichen Speicherverwaltung sind insbesondere zwei häufig auftretende Fehlertypen verbunden:

Das Speicherleck Von einem Speicherleck (memory leak) spricht man dann, wenn ein Programm im Laufe der Zeit immer mehr Speicher vom Betriebssystem anfordert, ohne dass es tatsächlich so viel Speicher für seine Aufgabe benötigt. Der Grund ist meist, dass vergessen wurde, Objekte bzw. Speicher, der nicht mehr benötigt wird, freizugeben und so dem allgemeinen Freispeicher-Pool wieder zuzuführen.

Die Hintergründe solcher Fehler sind sehr unterschiedlich, aber oft existieren gar keine Zeiger mehr in den zuviel belegten Speicher. Es gibt z.B. Funktionen, die Speicher belegen und Zeiger darauf zurückgeben. Manchmal bietet die Bibliothek, aus der eine solche Funktion stammt, eine Infrastruktur, die nicht mehr benötigten Speicher freigibt, weil er z.B. mit irgendeiner Resource verknüpft ist, deren Lebenszeit-Ende an einer anderen Stelle bekannt wird. Gibt es so etwas nicht, muss sich der Programmierer, der die Bibliothek verwendet, selbst darum kümmern, nicht mehr benötigten Speicher wieder freizugeben.

In vielen anderen Fällen ist es ein leicht vorkommender Flüchtigkeitsfehler des Programmierers, dass er Objekte anlegen lässt und sie vor Verlassen des Gültigkeitsbereichs des daraufzeigenden Zeigers nicht wieder freigibt. Es kommt auch vor, dass eine Zeigervariable, die noch einen Zeiger auf ein existierendes Objekt enthält, mit einem neuen Wert überschrieben wird, wodurch der Speicher des Objekts, auf das der Zeiger vorher gezeigt hat, nun auch verloren ist, falls nicht noch an einer anderen Stelle ein weiterer Zeiger auf das alte Objekt existiert.

Wie auch immer es zu einem Speicherleck gekommen ist; sobald dem Betriebssystem der Speicher ausgeht, führt es zur gewaltsamen Programmbeendigung durch das Betriebssystem. Es tötet dann der Reihe nach die größten Verschwender, um das System

wieder in einen bedienbaren Zustand zu versetzen. In aller Regel ist das verursachende Programm dabei das erste, welches getötet wird. Dabei gehen ungespeicherte Daten verloren.

Hängende Zeiger Hängende Zeiger (dangling pointers) sind sozusagen das Gegenteil zum Speicherleck. Während beim Speicherleck nicht mehr benötigter Speicher vergessen wurde und nicht freigegeben wurde, wurde beim hängenden Zeiger Speicher freigegeben, zu dem noch Zeiger existieren, die eventuell auch noch verwendet werden können. Diese Zeiger zeigen dann unter Umständen in Speicherbereiche, die von der Anwendung gar nicht mehr belegt werden. In diesem Fall führt ein Zugriff über einen hängenden Zeiger zu einem Speicherzugriffsfehler (Segfault) und oft in dessen Folge zum sofortigen Programmabsturz, bei dem nicht gespeicherte Daten verloren gehen.

Viel unauffälliger, dafür aber umso schlimmer, äußert sich ein hängender Zeiger wenn er zufällig in einen Speicherbereich zeigt, der von der Anwendung noch belegt ist, aber inzwischen von ganz anderen Objekten verwendet wird. Dann nämlich führt ein schreibender Zugriff über den Zeiger zu einer mehr oder weniger zufälligen Korruption der dort gespeicherten Daten. Ein lesender Zugriff liefert ungültige Daten im Sinne des vermeintlich dort gepeicherten Objekts zurück und führt somit wahrscheinlich auch zu einer Datenkorruption oder mindestens zu einem Anzeigefehler oder anderen weniger schlimmen Fehlfunktionen. Die Anwendung stürtzt in diesem Fall nicht notwendig sofort ab, was bedeutet, dass z.B. beim nächsten Abspeichern der Daten in eine Datei falsche Daten auf die Festplatte geschrieben werden und somit die Daten endgültig zerstört sind.

2.2 Lösung durch automatische Speicherverwaltung

Die Lösung dieser Probleme liegt darin, dem Programmierer die Aufgabe der Speicherverwaltung durch ausgeklügelte Algorithmen abzunehmen. Das reduziert die Häufigkeit der genannten Probleme und macht den geschriebenen Code übersichtlicher. Es muss eine als ästhetisch und sinnvoll erachtete Strukturierung einer objektorientierten Anwendung nicht mehr wegen der Speicherverwaltung abgeändert werden, denn manchmal ist die Ursache für das Lebenszeitende eines Objekts nicht dort erkennbar, wo sich der Zeiger darauf befindet. Nach der herkömmlichen Methode müssten dann Hilfskonstruktionen oder Benachrichtigungskanäle geschaffen werden um seine Freigabe zu veranlassen. Dieser Aufwand entfällt also, wenn die Speicherverwaltung automatisch vorgenommen wird. Letztlich verringert das die Entwicklungszeit und somit auch die Entwicklungskosten.

Das gefährliche Problem mit den baumelden Zeigern lässt sich damit komplett eliminieren, mit den Speicherlecks sieht es jedoch leider nicht ganz so gut aus, da es neben den genannten Möglichkeiten noch andere Formen von Speicherlecks gibt, die sich nach momentanem Erkenntnisstand noch nicht sinnvoll algorithmisch erkennen lassen. Dabei geht es um Daten, zu denen noch erreichbare Zeiger existieren, die aber von der Programmsemantik her trotzdem nicht mehr gebraucht werden. Das zu erkennen würde erfordern, alle möglichen zukünftigen Läufe des Programms vorauszuberechnen, was nicht leistbar ist. Deshalb ist es sogar trotz des Einsatzes einer automatischen Speicherverwaltung manchmal nötig, eine Art von Freigabeoperation durchzuführen. Sie besteht

darin einen Zeiger auf Null zu setzen, um damit deutlich zu machen, dass die von dort aus erreichbaren Daten von einem Codefragment aus gesehen nicht mehr benötigt werden.

Die automatische Speicherverwaltung bringt auch teilweise Performance-Nachteile mit sich, die jedoch stark von den verwendeten Algorithmen sowohl im Programm als auch in der Garbage-Collection selbst abhängen, wie wir im nächsten Abschnitt sehen werden.

Das ganz große Streitthema in Bezug auf Garbage-Collection ist jedoch die deterministische Finalisierung, denn wenn ein im Hintergrund wartender Garbage-Collector nur gelegentlich aktiv wird, weil es sich nicht lohnt, ständig aktiv zu sein, dann werden Objekte unter Umständen erst lange Zeit nach ihrer herkömmlichen Freigabe von der Garbage-Collection entsorgt. Wird dann der Aufruf des Destruktors an den Garbage-Collector gekoppelt, dann werden knappe Resourcen, die von einem Objekt belegt wurden und typischerweise vom Destruktor freigegeben werden, erst viel später freigegeben als sie es nach der herkömmlichen Speicherverwaltung geworden wären. Das bedeutet, dass die Objekte für die Freigabe knapper Resourcen oder ähnlicher deterministisch auszuführender Finalisierungsoperationen zusätzliche Methoden anbieten müssen, die dann vom Benutzer der Objekte statt einer expiliziten Destruktion aufgerufen werden müssen. D.h. statt eines **delete** Aufrufs steht dann im Code ein Aufruf der Finalisierungsfunktion, womit sich die Grundlage der Argumentation für automatische Speicherverwaltung zu relativieren beginnt. Vergisst der Programmierer nämlich den Aufruf des Finalisierers, dann werden knappe Resourcen und andere Aufräumarbeiten erst bei der Destruktion durch den Garbage-Collector, also viel später ausgeführt, womit wir nicht viel besser dran sind als vorher. Bei den meisten Programmen wird ein Garbage-Collector jedoch in relativ regelmäßigen Abständen aktiv und somit würden die Aufräumarbeiten bei Vergessen des Finalisierer-Aufrufs doch noch recht zeitnah erfolgen, nur eben nicht deterministisch.

2.2.1 Speicherkompaktierung

Ein weiterer Vorteil von automatischer Speicherverwaltung kann die effizientere Ausnutzung des Speichers sein.

Man kennt das Problem von Festplatten: Wenn eine Festplatte längere Zeit in Benutzung war, dann sind durch die Erstellung und Löschung oder die Vergrößerung und Verkleinerung von Dateien Lücken zwischen den Dateien entstanden, die vielleicht gar nicht groß genug sind um eine neue Datei dort anzulegen. Die neue Datei muss dann zerstückelt abgespeichert werden, was aber wesentlich langsamer ist, weil der Kopf der Festplatte ständig neu positioniert werden muss.

Ein ähnliches, teilweise noch stärkeres Problem existiert beim Hauptspeicher ebenso. Die zusätzliche Schwierigkeit hierbei ist, dass zumindest im virtuellen Adressraum eines Programms sämtliche Objekte immer als zusammenhängender, fortlaufend adressierter Block vorliegen müssen, da die Feldbezeichner bei der Compilierung eines Programms zu festen Offset-Adressen (Zahlen) umcodiert werden, die zur jeweiligen Basisadresse eines Objekts hinzuaddiert werden. Es ist deshalb nicht möglich, ein Objekt in mehrere Teile zu zerlegen, wie es bei Dateien auf der Festplatte möglich ist. Hinzu kommt das Problem, dass ein Programm den von ihm belegten Speicher nur am Ende vergrößern oder verkleinern kann. Es kann also vorkommen, dass in einem Programm, das einmal

viel Speicher benötigt hatte, ein kleines Objekt im hinteren Bereich des Speichers verhindert, dass der ungenutzte Speicher wieder freigegeben werden kann. Es wäre also wünschenswert, wenn in regelmäßigen Abständen sämtliche Objekte kompakt an den Anfang des Speichers zusammengeschoben und anschließend am Ende freigewordener Speicher freigegeben werden würde.

Ein weiterer Bonus ist, dass durch die dichter beieinander liegenden Objekte der stark begrenzte Prozessorcache besser ausgenutzt wird, denn wenn eine Seite eingecached wird, dann sind bei kompakter Verwendung des Speichers nur aktive Objekte darin enthalten und nicht auch inaktive. Es wird deshalb wahrscheinlicher, dass sich der Prozessor einen Speicherzugriff sparen kann und stattdessen ein Objekt aus dem Cache verwenden kann.

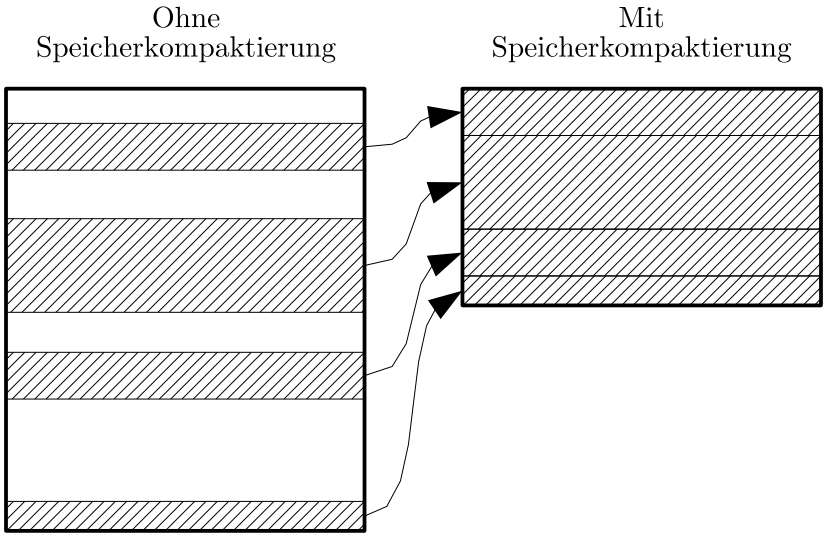

Abbildung 1: Effekt der Speicherkompaktierung

Leider ist so eine Technik nicht ohne Nachteil, denn durch das Verschieben der Objekte im Speicher wird es erforderlich, eine zusätzliche Indirektion bei Zeigern auf solche verschiebbaren Objekte in Kauf zu nehmen. Die Zeiger und Referenzen in C++ enthalten nämlich feste Zahlenwerte. Diese alle zu aktualisieren, wenn die Objekte im Speicher verschoben werden, ist nicht möglich, da durch Umcasten ein solcher Zeiger innerhalb jeden beliebigen Datentyps abgelegt sein kann, und beliebige Daten zu verändern, die wie Zeiger aussehen, wäre zu riskant. Es muss also ein nicht verschiebbares Zwischenobjekt herhalten, das einen Index in eine ebenfalls nicht verschiebbare Tabelle enthält, wo der eigentliche Zeiger abgelegt ist. Bei der Kompaktierung wird die Anwendung angehalten und die Einträge in der Tabelle entsprechend der neuen Positionen der Objekte aktualisiert. Das Anhalten stellt dabei einen weiteren Nachteil dar, doch würde man das nicht machen, könnte es passieren, dass die Anwendung nach dem Kopieren eines Objekts über einen veralteten Zeiger auf dieses Objekt schreibend zugreift und anschließend der Zeiger auf die neue Position umgebogen wird, wodurch die Veränderung praktisch ungeschehen gemacht wird. Auch könnte es passieren, dass ein Objekt wärend des Kopierens ver-

ändert wird, was ebenfalls zur Datenkorruption führen würde. Es muss also mindestens das Schreiben über einen Zeiger, dessen Objekt gerade kopiert wird, aufgehalten werden, bis das Objekt fertig kopiert ist und die neue Position in der Tabelle eingetragen wurde. Wird der Garbage-Collector nicht als Hintergrundprozess sondern als Nebeneffekt bestimmter Speicherverwaltungsaufrufe wie z.B. dem Operator **new** aufgerufen, dann könnten wir bei Single-Thread-Programmen auf ein Locking verzichten, da uns dann prinzipiell nichts dazwischen kommen kann, doch immer mehr Programme setzen auf Multi-Threading, also kommen wir nicht drumherum. Denn während der eine Thread im Zuge eines **new**-Aufrufs den Speicher kompaktiert, könnte der andere im geteilten Speicherbereich schreibend zugreifen, was verhindert werden muss.

3 Garbage-Collection Algorithmen

In diesem Abschnitt geht es darum, wie ein Garbage-Collector herausfinden kann, welche Objekte noch benutzt werden und welche Abfall sind. Die Tatsache, dass ein Objekt noch benutzt wird, ist aus seiner Sicht so definiert, dass im Referenzbaum einen Weg von einer aktiven Referenz, wie einer globalen oder lokalen Zeiger- oder Referenzvariable zu dem Zielobjekt existiert. Existiert kein solcher Weg, dann geht der Collector davon aus, dass das Objekt nicht mehr benutzt, weil nicht mehr aufgefunden, werden kann.

Eine andere Aufgabe ist die möglichst effiziente Freigabe des Abfalls. Am besten wäre es dabei, wenn man die überflüssig gewordenen Objekte überhaupt nicht mehr im Einzelnen anfassen müsste und sich nur mit den aktiven befassen könnte, doch muss in einem C++ Programm vor der Freigabe des Speichers noch der Destruktor aller freizugebenden Objekte aufgerufen werden. Die letztendliche Freigabe des Speichers kann jedoch vom Garbage-Collector optimiert werden.

Um bei der Speicherverwaltung optimale Ergebnisse zu erzielen, müsste dem Garbage-Collector entweder eine Regel für jedes Objekt angegeben werden, nach der er es behandeln soll, oder er muss versuchen zu erahnen, welche Methode für ein Objekt am besten geeignet ist. Viele der einfachen Garbage-Collectoren machen hier allerdings keine Unterscheidung und behandeln alle Objekte nach dem gleichen Prinzip.

3.1 Referenzzählung

Die naheliegendste Methode, herauszufinden, wie viele Referenzen noch auf ein Objekt existieren, ist mitzuzählen. Das erfordert allerdings, dass der Compiler bei jeder Zuweisung an eine Referenz entsprechenden zusätzlichen Code generieren muss, der den Zähler hochzählt und ihn wieder herunterzählt, wenn die Referenz freigegeben wird. Das kann entweder geschehen, weil ihr ein Zeiger auf ein anderes Objekt zugewiesen wird oder weil der Block, in dem sie deklariert ist, verlassen wird.

Geht eben genannter Zähler bei einer dieser Dekrementierungen auf Null, so bedeutet das, dass von keinem Objekt, dessen eigener Zähler größer als Null ist, mehr auf dieses Objekt verwiesen wird. Damit ist es für diesen Algorithmus unerreichbar und nach der Definition Müll. Der vom Objekt belegte Speicher kann also nun freigegeben werden

und alle Referenzen, die das Objekt selbst enthalten hat müssen nun ebenfalls heruntergezählt werden, wodurch weitere Objekte unerreichbar werden können und ebenfalls freigegeben werden müssen. Das ist ein rekursiver Prozess, bei dem jedes Müll-Objekt besucht werden muss, also kann auch gleich der Destruktor aufgerufen werden. Das hat den Vorteil, dass die Finalisierung bei dieser Methode sofort nach dem Absinken des Referenzzählers auf Null durchgeführt wird. Referenzzählung beisitzt damit anders als die nachfolgenden Algorithmen die gewünschte deterministische Finalisierung ohne zusätzliche Finalisierungsfunktionen.

Leider ist dieser Algorithmus nur für azyklische Referenzbäume und somit nicht für einen allgemein verwendbaren Garbage-Collector geeignet, da jeder Verweis auf ein Objekt die Freigabe des Objekts verhindert, gleichgültig ob das auf es zeigende Objekt von einer globalen oder lokalen Variable aus erreichbar ist oder nicht. Somit würden Objekte, die direkt oder indirekt aufeinander verweisen niemals freigegeben, was zum Speicherleck führen würde.

Die nachfolgende Abbildung zeigt schematisch eine Situation bei Anwendung von Referenzzählung. Bei einem Objekt ist gerade der Zähler auf Null gefallen. Es wird deshalb demnächst entfernt. Vom Stack sehen wir einen Zeiger auf ein Objekt, das wiederum auf ein Zykel auf drei anderen Objekten verweist. Geht jetzt die Zeigervariable auf dem Stack aus ihrem Gültigkeitsbereich, dann wird nur das eine direkt erreichbare Objekt freigegeben werden. Die drei anderen bleiben unnerreichbar im Speicher liegen und verschwenden unnötig Platz.

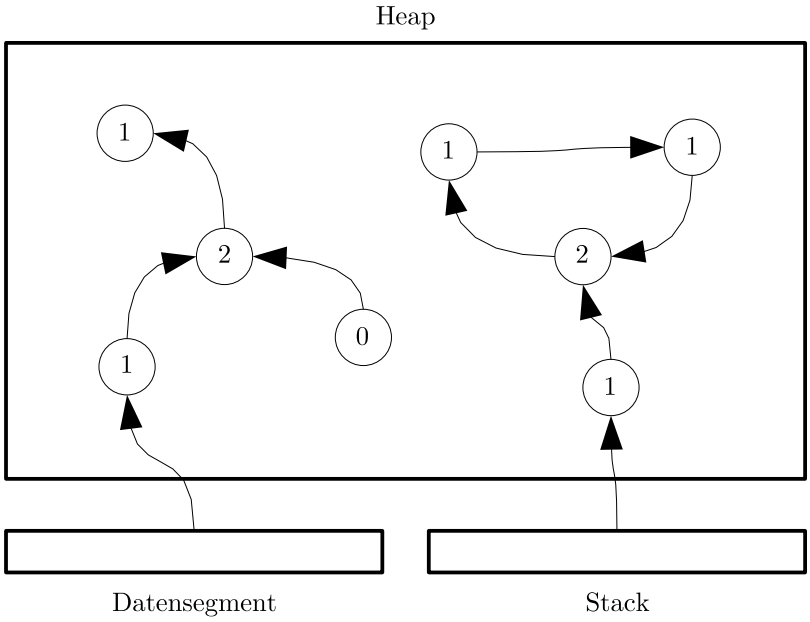

Abbildung 2: Referenzzählung

Um diesem Problem zu begegnen gibt es in den Bibliotheken, die Referenzzählung anbieten spezielle Zeiger, die den Zähler nicht beeinflussen. Sie verweisen dann zwar auf ein Objekt, verhindern aber nicht seine Freigabe. Wird ein so referenziertes Objekt dann freigegeben, geht dieser Spezial-Zeiger auf Null, was vor jeder Verwendung abgefragt werden müsste. Weil das aber Fehleranfällig wäre, geht man z.B. bei der Smart-Pointer-Bibliothek von Boost so vor, dass ein solcher Spezial-Zeiger vor einer Verwendung immer erst in einen normalen Referenzzählungs-Zeiger umgewandelt werden muss. Dabei wird überprüft, ob das Objekt noch existiert und gegebenenfalls eine Ausnahme ausgelöst, die man dann abfangen kann, um diesen Fall zu behandeln.

Bei reinen Datenobjekten, die selbst keine Referenzen auf andere Objekte enthalten, ist dieser Algorithmus aber gerade wegen seiner sehr einfachen Implementation und der impliziten deterministische Finalisierung ein probates Mittel, um unnötige Kopien von großen Datenmengen zu vermeiden. So nutzen z.B. manche Implementationen der Klasse std :: string intern Referenzzählung um den Nutzer von der Notwendigkeit der Freigabe der Zeichen-Daten zu befreien und trotzdem unnötige Kopien zu vermeiden. Erst wenn eine „Kopie“ verändert wird, werden tatsächlich die hinterlegten Zeichendaten kopiert. (copy on write – COW).

3.2 Verfolgende Kollektoren

Dem angesprochenen Problem mit teilweise zyklischen Referenzgraphen wird in der Praxis mit einem Verfolgungsverfahren begegnet. Es wird regelmäßig im Hintergrund oder als Nebeneffekt einer Speicherverwaltungsfunktion ein Suchlauf durchgeführt. Dabei werden zuerst alle existierenden dynamisch angelegten Objekte als unerreichbar markiert. Danach werden ausgehend von den globalen Variablen und denen in aktiven Stack-Frames alle Zeiger und Referenzen rekursiv besucht und deren Zielobjekte, falls es sich um dynamisch angelegte Objekte handelt, als erreichbar markiert. Stack-Objekte und globale Variablen sind ohnehin erreichbar, da bei ihnen die Suche beginnt. Trifft der Suchprozess auf ein Blatt im Referenzgraph oder auf ein bereits als erreichbar markiertes Objekt, so wird von dort aus nicht weitergesucht. Dadurch wird verhindert, dass Zyklen zu einer Endlosschleife führen, worum es ja insbesondere bei diesem Verfahren geht.

Neue Objekte werden, wenn sie neu angelegt werden, immer als erreichbar markiert. So wird verhindert, dass ein neues Objekt fälschlicherweise freigegeben wird, wenn es während des Suchprozesses angelegt wird. Der Suchprozess ist also unterbrechbar und erfordert nur minimales Locking der einzelnen besuchten Referenzen.

Dieser Suchprozess tritt allerdings meist zu relativ zufälligen Zeitpunkten auf und verbraucht bei großen Applikationen mit vielen lebenden Objekten auch nicht unerheblich viel Zeit. Auch muss der komplette Suchprozess jedesmal von Null anfangen und kann sich keine gespeicherten Daten zu Nutze machen.

Je nach dem in welchen Zeitabständen ein solcher Kollektor aktiv wird, bleiben eine gewisse Anzahl von unerreichbaren Objekten immer unzerstört im Speicher liegen, bis sie freigegeben werden. Das verbraucht gegenüber einer manuellen Zerstörung zusätzlichen Speicher.

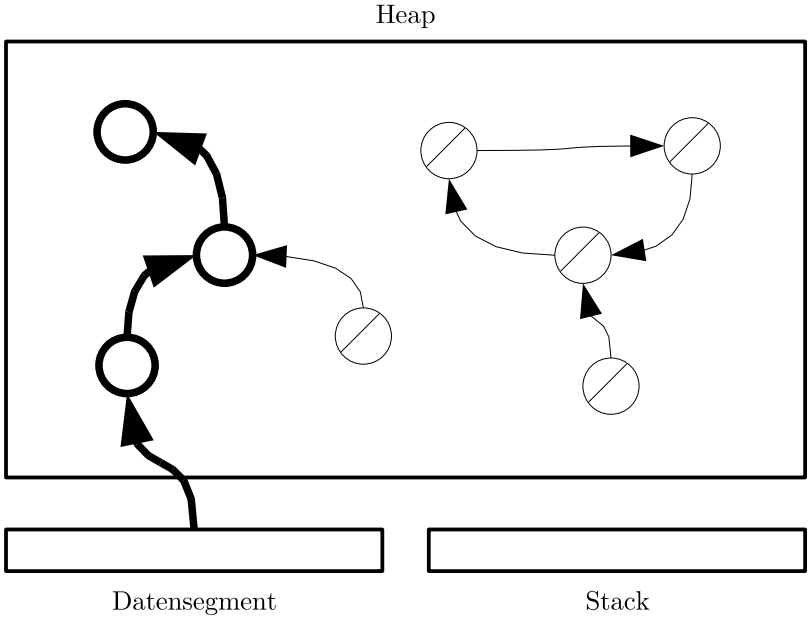

Abbildung 3: Verfolgungsprozess

Da es sich bei diesem Ansatz grundsätzlich um eine funktionierende Lösung handelt, weil insbesondere keine unerreichbaren Objekte ewig unzerstört bleiben, gibt es von ihm in anderen Sprachen wie z.B. Lisp verschiedene tatsächlich realisierte Implementationen mit unterschiedlichen Eigenschaften. Bei C++ sieht es allerdings nach dem aktuellen Standard so aus, dass sich ein solcher Kollektor nicht in einer „exakten“ Variante implementieren lässt, da die Sprache C++ keine Reflexivität bietet und sich Zeiger auch als Zahlen abspeichern lassen. Ein verfolgender Kollektor muss deshalb alles als Zeiger berücksichtigen, was auch nur annähernd so aussieht. Dadurch kann es aber passieren, dass einige Objekte fälschlicherweise als erreichbar markiert werden, weil zufällig irgendwelche anderen „erreichbaren“ Daten wie ein Zeiger zu ihnen aussahen. Diese Fehleinschätzung birgt auch das Risiko einer Denial-Of-Service Attacke, indem über einen beliebigen Kanal dem Programm Datenmüll zugespielt wird, der viele verschiedene plausible „Zeiger“ enthält. Dadurch würden eine Menge Objekte unzerstört bleiben und eventuell die Anwendung aus Speichermangel abstürzen. Das Risiko eines solchen Angriffs wird zwar als gering eingestuft, denkbar ist es jedoch. Deshalb muss ein solcher Kollektor eine Möglichkeit bieten fremde Daten als „Zeigerfrei“ zu deklarieren. In ihnen wird dann nicht nach Referenzen gesucht. Das verkompliziert die Verwendung von automatischer Speicherverwaltung allerdings zusätzlich.

3.2.1 „mark and sweep“

Die einfachste Form eines verfolgenden Kollektors ist der sogenannte „mark and sweep“-Kollektor. Dabei werden nach dem Suchvorgang, der „mark“-Phase, in der sogeannten

„sweep“-Phase sämtliche existierenden Objekte durchgegangen und alle zuvor nicht als erreichbar markierten freigegeben. Also wird in der „mark“-Phase jedes erreichbare und in der „sweep“-Phase jedes unerreichbare Objekt besucht und somit alle Seiten des Programmspeichers benötigt. Diese müssen dann gegebenenfalls von der Festplatte eingelesen werden, was zusätzlich bremst. Sehr einfache Implementationen unterbrechen zudem den Programmfluss für die Dauer des gesamten Vorgangs und da dieser Vorgang aus den eben beschriebenen Gründen sehr lange dauert, stellt ein solcher Kollektor eine starke Performance-Bremse dar. Er ist langsamer als manuelle Speicherbereinigung und verbraucht viel zusätzlichen Speicher für eigentlich unerreichbare Objekte, da er wegen der schlechten Performance nicht oft ausgeführt werden kann.

Eine erweiterte Variante der „mark and sweep“-Technik koppelte die „Freigabe“ des Speichers an die Speicherallokationsfunktion. Der Speicher wird nach dem Aufruf des Destruktors nur noch freigegeben, wenn die Anwendung deutlich mehr Speicher belegt, als sie benötigt, oder die Speicherresourcen systemweit knapp sind. Ansonsten wird bei der Allokation neuen Speichers einfach nicht genutzter Speicher, der bereits alloziiert ist, an ein neues Objekt vergeben. Der Kollektor reduziert damit die Betriebssystem-Interaktionen, die wegen des nötigen Kontext-Wechsels vom Usermode in den Kernelmode und zurück zusätzlich Zeit verbrauchen. Auch entlastet dieses Verfahren die Speicherverwaltung des Betriebssystems. Allerdings wird dadurch wertvoller, weil schneller RAM-Speicher, verschwendet, der zum Cachen von langsamen Block-Devices wie Festplatten verwendet werden könnte.

Diese Variante wird von dem als C/C++-Bibliothek vorliegenden Garbage-Collector von Boehm, Demers und Weiser verwendet.

3.2.2 Kopierkollektoren

Eine weitere Form des Verfolgenden Kollektors ist der Kopierkollektor. Bei ihm wird genauso wie bei „mark and sweep“ ein Suchlauf durchgeführt, bei dem alle erreichbaren Objekte markiert werden. Der Unterschied ist aber, dass der Speicher bei Kopierkollektoren in mindestens ein Paar aus zwei getrennten Speicherbereichen aufgeteilt wird. Nach der „mark“-Phase werden dann in einer gut unterbrechbaren Kopierphase alle erreichbaren Objekte aus dem einen Speicherbereich in den anderen kopiert und dabei kompakt dort abgelegt. Das hat besonders bei Sprachen ohne Destruktor wie z.B. Java den Vorteil, dass nur aktive Objekte überhaupt angefasst werden müssen. Bei C++ müsste aber der Desktruktor aufgerufen werden. Die Freigabe des Speichers unerreichbarer Objekte im Einzelnen entfällt allerdings. Nach der Kopierphase und der einhergehenden Kompaktierung kann am Ende der beiden Speicherbereiche überflüssig gewordener Speicher in einem vernünftigen Maß freigegeben werden, so dass nicht bei der nächsten Allokation der Speicher wieder vergrößert werden muss. Vorraussetzung für die effiziente Implementation eines Kopierkollektors ist die vom Betriebssystem abhängige Möglichkeit unabhängige Speicherbereiche zu verwalten. Denn ansonsten ist eine Vergrößerung oder Verkleinerung des Speichers nur möglich, wenn gerade der erste Speicherbereich aktiv ist, was im Falle der Vergrößerung wegen Allokation bedeuten könnte, dass um eine einzelne Allokation durchzuführen es passieren kann, dass alle Objekte spontan aus dem

zweiten in den ersten Speicherbereich umkopiert werden müssten, was für die Allokation eine unvorhersehbare Latenz mit sich bringen würde. Linux bietet diese Möglichkeit unabhängige Speicherbereiche zu verwalten im Form von anonymen Mappings, andere aktuelle Betriebssysteme sicherlich auch, so dass das nur in Ausnahmefällen ein echtes Problem darstellt.

Ein weiterer großer Vorteil von Kopierkollektoren ist die besonders einfache Allokationsstrategie: Neue Objekte werden einfach am Ende des aktiven Bereichs oder während der Kopierphase am Ende des Zielbereichs angelegt. Die bei der herkömmlichen Speicherwaltung erforderliche aufwendige Suche mit Hilfe von Bitmaps nach einem genügend großen zusammenhängenden Block entfällt völlig. Das spart Speicherplatz, weil keine Bitmaps mehr benötigt werden, und viel Zeit.

Ein Kopierkollektor verbraucht allerdings immer mindestens doppelt so viel Speicherplatz, wie von den aktiven Objekten benötigt wird. Durch die Speicherlücken, die bei der herkömmlichen Speicherverwaltung auftreten, kann in ungünstigen Fällen aber mehr als dieses doppelte an Speicherplatz verbraucht werden, somit ist die Verschwendung beim Kopierkollektor zwar groß aber in festen Grenzen. Diese große Verschwendung wirkt sich insbesondere bei datenintensiven Programmen, wie z.B. Bildbearbeitungssoftware oder Computerspielen, sehr negativ aus, die dann für ihre großen Daten von der Garbage-Collection unabhängige Verfahren enthalten müssen, um weniger Speicher zu verschwenden. Die besondere Verwaltung von sehr großen Datenmengen ist aber nichts Neues. Die großen Bildbearbeitungsprogramme wie Adobe Photoshop oder Corel Photopaint enthalten seit jeher ihre eigene Speicherverwaltung inklusive eigenem Swap-System, um überhaupt eine annehmbare Performance zu erreichen. Die Glibc alloziiert größere Speicherblöcke automatisch in Form von anonymen Mappings, so dass bei ihrer Freigabe der Speicher auch garantiert an das Betriebssystem zurückgegeben werden kann. Etwas Ähnliches könnte der Kopierkollektor ebenfalls tun, um dem Problem zu begegnen.

3.2.3 Generationelle Kollektoren

Die Sonderbehandlung von Objekten abhängig von ihrem Verwendungszweck hat also offensichtlich große Vorteile. In die gleiche Richtung geht ein Optimierungsmechanismus, der Objekte abhängig von ihrer bisherigen Lebenszeit unterschiedlich behandelt.

Dabei werden ältere Objekte seltener durchsucht als neuere. Das ist günstig, weil ältere Objekte mit einer höheren Wahrscheinlichkeit auch noch länger existieren werden. Man nehme z.B. die Objekte rund um das Applikations-Hauptfenster einer grafischen Anwendung. Das Fenster existiert für die komplette Dauer der Programmausführung. Es macht also keinen Sinn ständig nachzuschauen, ob vielleicht die Menüzeile gelöscht wurde. Ähnlich verhält es sich bei vielen Objekten, deshalb bringt eine seltenere Durchsuchung älterer Objekte mehr Vorteile als Nachteile. Es wird wenig zusätzlicher Speicher verbraucht, aber viel Zeit gespart.

Bei Kopierkollektoren bedeutet diese Optimierung, dass mehrere Paare von Speicherbereichen für die verschiedenen Generationsklassen benötigt werden. Bei „mark and sweep“-Kollektoren betrifft diese Optimierung lediglich den Suchprozess und ist somit auch in C++ implementierbar. Der Boehm-Demers-Weiser-Kollektor macht ebenfalls

davon Gebrauch.

4 Ergänzung der Kernsprache

Wie wir gesehen haben ist die bisher bekannte performanteste Lösung ein Kopierkollektor mit generationeller Optimierung. Dieses Verfahren wird deshalb auch von den neueren JAVA-Versionen eingesetzt. JAVA hat es in diesem Punkt allerdings wesentlich leichter: JAVA wurde von vorneherein mit Garbage-Collection im Sinn entworfen. Die entscheidenden Vorteile sind: Zeiger sind opak, d.h. nicht in Zahlen umcastbar und erst recht nicht zurück, und die Klassen besitzen keine Destruktoren, sondern stattdessen Finalisierungsfunktionen. JAVA besitzt zudem Reflexivität, das bedeutet JAVA-Programme können auf ihrer eigenen internen Struktur der Objekte arbeiten. Ein Algorithmus, der rekursiv alle Zeiger absucht ist dort leicht, sogar innerhalb der Sprache selbst, implementierbar. C++ bietet das alles nicht. Für die Implementation eines exakten verfolgenden Kollektors ist es erforderlich, zu jedem Objekt zu erfassen, welche Referenzen es besitzt und wo diese hinzeigen. Eine Möglichkeit wäre, eine spezielle Basisklasse einzuführen, die dem Kollektor Zugriff auf eine solche Liste gewährt. Die Zeiger müssten dann zusätzlich alle dort registriert werden. Vergisst man das, funktioniert der Kollektor nicht richtig und löscht eigentlich noch benötigte Objekte. Ein gefährlicher baumelnder Zeiger wäre die Folge.

Ein weiteres Problem ist die Tatsache, dass die Zeiger tatsächliche Speicheradressen enthalten. Ein Kopierkollektor ist deshalb damit nicht realisierbar. Es wäre also zusätzlich ein neuer Zeigertyp notwendig, der die erforderliche Indirektion enthält.

Mindestens das erste Problem erfordert, um es sicher zu machen, die Unterstützung des Compilers. Soll dieser doch die erforderliche Liste anlegen. Eine zusätzliche Basisklasse ist dann auch nicht mehr erforderlich, da die Hilfsdatenstruktur unabhängig von den betreuten Objekten angelegt werden kann. Damit der Compiler die neuen Zeiger eindeutig erkennen kann, wäre entweder ein neues Schlüsselwort erforderlich oder eine angepasste Symbolik für die Deklaration von Zeigern zu verwalteten Objekten.

4.1 Microsofts Vorschlag

„Das soll es sein“ dachten sich auch die Entwickler bei Microsoft als sie die neue Sprache C++/CLI entwarfen, die im Rahmen des „Common Language Interface“ Garbage-Collection unterstützt wie die anderen CLI-Sprachen auch.

Dort gibt es neben den herkömmlichen Zeigern und Referenzen sogenannte Handles, die statt mit einem Stern mit einem Dach deklariert werden. Diese Handles lassen sich zwar auslesen, liefern dann allerdings den Index in der Verwaltungstabelle. Es existiert auch eine Methode um die tatsächliche Adresse zu ermitteln, nur sollte man davon keinen herkömmlichen Zeiger konstruieren, da die verwalteten Objekte plötzlich von der Speicherverwaltung verschoben werden könnten.

Microsoft macht mit diesem Vorstoß nicht nur vor, wie es gehen kann, sondern setzt auch das Standardisierungsgremium stark unter Druck, denn viele .NET Programme

werden heute schon in C++/CLI entwickelt, was bedeutet, dass sich die Microsoftsche Erweiterung zu einer Art Defacto-Standard entwickeln kann.

Verwaltbare Klassen werden in C++/CLI mit einem zusätzlichen Schlüsselwort **gc** markiert. Das lässt vermuten, dass Microsoft die Tabelle mit den enthaltenen Referenzen in den Objekten selbst unterbringt, was nicht unbedingt notwendig wäre aber praktisch ist, da dann keine separate Baumstruktur verwaltet werden muss.

Zur Erzeugung dynamischer verwalteter Objekte wird dort der Operator **gcnew** eingesetzt. Die Unterscheidung ist notwendig, da von verwaltbaren Objekten auch normale dynamische und auch Stack-Objekte angelegt werden können. Die RAII-Technik zur automatischen Speicherverwaltung mittels auto-Objekten ist damit in C++/CLI weiterhin möglich, was ein weiteres Plus für eine sanfte Migration zur neuen Version darstellt. Die herkömmlichen Mechanismen funktionieren allesamt nach wie vor, wie man es erwartet (und bergen natürlich auch die herkömmlichen Risiken).

5 Offene Probleme

Wenn das alles so gut funktioniert, warum ist es dann nicht schon längst neuer Standard? Trotz all der Vorteile gibt es ein paar ganz entscheidende Nachteile und entsprechende Kritik, von Leuten, die davon sehr betroffen wären.

5.1 Deterministische Finalisierung

Die deterministische Finalisierung ist auch beim Microsoftschen Vorstoß nicht implizit. Dort behilft man sich (soweit ich es den kleineren Beispielprogrammen im Internet entnehmen konnte) mit einem expliziten Aufruf des Destruktors über seinen Namen. Die Objekte müssen also damit rechnen, dass ihr Destruktor mehrfach aufgerufen werden kann und dass andere Funktionen nach dem Destruktor noch aufgerufen werden könnten. Die VMT (virtual method table) wird bei CLI anders gehandhabt, so dass auch nach dem Destruktor noch die überschriebenen Versionen der abgeleiteten Klasse aufgerufen werden. Diese müssen dann aber damit rechnen, dass das Objekt schon teilweise finalisiert ist.

Insgesamt scheint mir der explizite Aufruf des Destruktors verwirrender als ein explizites **delete**. Separate Finalisierungsfunktionen kann man natürlich deklarieren. Diese müssten dann aber auch vom Destruktor aufgerufen werden, um dem RAII-Prinzip zu dienen. Damit kann also sowohl der Destruktor als auch ein separater Finalisierer mehrfach aufgerufen werden. Ich finde, das verkompliziert die Entwicklung von neuen Klassen aber auch insbesondere die Migration bestehenden Codes doch erheblich.

Das Sprachkonzept hinter den C++-Klassen zeichnete sich aber bisher dadurch aus, dass es ganz besonders einfach und leichtgewichtig ist. Es gibt z.B. keine zwingende Basisklasse und sogar Hardware-nahe Datenstrukturen lassen sich gut in entsprechend konstruierte C++-Klassen umcasten. Es wäre schade, wenn diese Vorteile einer Garbage-Collection ohne nennenswerten programmierpraktischen Vorteil zum Opfer fielen.

5.2 Performance

Nicht vorhersehbare Unterbrechungen des Programmflusses sind in manchen Softwarebereichen nicht tolerierbar. Deshalb muss ein Garbage-Collector die Möglichkeit bieten, vom Anwendungsprogramm an einem genau ausgewiesenen Zeitpunkt gestartet zu werden und dabei vorgegebene Zeitlimits nicht zu überschreiten. Das verkompliziert die Entwicklung des Garbage-Collectors selbst, aber auch die Entwicklung der Anwendungsprogramme.

Vielen Entwicklern ist zudem ein allgemeiner Garbage-Collector zu langsam. Diese Meinung rührt allerdings teilweise daher, dass die frühen JAVA Garbage-Collectoren auf „mark and sweep"-Basis tatsächlich sehr langsam waren. Inzwischen ist es in vielen Fällen so, dass die ausgereifteren aktuellen Garbage-Collectoren sogar schneller sind, als die herkömmliche explizite Speicherverwaltung. Eine speziell auf den Anwendungsfall zuschnittene eigene Speicherverwaltung, sofern sie wirklich gut gemacht ist, sticht jedoch nach wie vor beide Standard-Methoden aus.

Eine integrierte Sonderbehandlung von großen Datenmengen beim Kopierkollektor ist allerdings praktisch ein Muss, jedoch nicht komplizierter als die existierende Sonderbehandlung.

Lässt man den Garbage-Collector gezielt während sonst inaktiver Zeiten laufen, sollte der Verlust an Performance kaum messbar sein, der Speicherverbrauch kann dadurch schon eher steigen.

An dieser Stelle sei noch einmal bemerkt, dass der Garbage-Collector nicht unbedingt einen Zusatzaufwand gegenüber der herkömmlichen Speicherverwaltung darstellt. Er ersetzt sie. Es hängt von den Algorithmen im Collector und im Programm ab, ob dadurch in Summe mehr oder sogar weniger Rechenzeit verbraucht wird.

5.3 Optionalität

Die Entwickler mancher Sofwarebereiche sind von einer Lösung mittels Garbage-Collector allerdings überhaupt nicht zu überzeugen. Insbesondere bei Echtzeit-Anwendungen möchte man die Kontrolle über das Programm nicht an eine fremde Macht abgeben. Für solche Fälle müsste es seitens des Sprachstandards eine Möglichkeit geben, den Collector vollständig zu deaktivieren. Natürlich müsste trotzdem noch alles möglich sein.

Existiert der Garbage-Collector erst einmal, dann wird er natürlich auch genutzt. Es ist auch damit zu rechnen, dass er gerade von Bibliotheks-Entwicklern, die meist mit den komplizierteren Referenzgraphen zu tun haben als die Anwendungsentwickler, gerne genutzt wird. Eine Bibliothek, die den Einsatz des Garbage-Collectors erzwingt, kann dann von einem Enwickler, der ihn aus anderen Gründen nicht verwenden will, natürlich nicht verwendet werden. Die Bibliothek müsste dann andernfalls auch in einer Version vorliegen, die den Garbage-Collector nicht benötigt, womit natürlich darüber diskutiert werden kann, ob man dann überhaupt die Version mit Garbage-Collector noch braucht. Das bedeutet also, dass einige verbreitete Bibliotheken in doppelter Ausführung weiterentwickelt werden müssen, wenn nicht noch irgendetwas erfunden wird, was die gewünschte Optionalität irgendwie mitbringt.

6 Fazit

Das Beste, was sich in Form einer Bibliothek innerhalb der bestehenden Sprache realisieren lässt, ist ein konservativer Garbage-Collector nach „mark and sweep“-Prinzip. Der Boehm-Demers-Weiser Garbage-Collector ist ein solcher. Er nutzt generationelle Methoden um die Performance zu verbessern, ist ständig unterbrechbar und angeblich auch gut verwendbar um ihn nachträglich in bestehende Projekte zu integrieren.

Die besseren Techniken sind leider noch nicht in einer einigermaßen sicheren Variante implementierbar. Meiner Meinung nach würde es an sich reichen, Reflexivität in der Sprache zu ermöglichen, dann könnte man mit Hilfe einer eigenen speziellen Zeigerklasse (Template) sogar einen exakten Kopierkollektor implementieren. Eine Sprachunterstützung in Form einer standardisierten Garbage-Collection wäre dazu gar nicht einmal nötig. Verschiedene Garbage-Collection Bibliotheken könnten dann um die Gunst der Anwender konkurrieren und würden so eventuell auch eher Lösungen für speziellere Zwecke bieten. Da Reflexivität auch zu anderen Zwecken, allen voran Testzwecke, hilfreich ist, ist, denke ich, das die beste Lösung.

Die Definition einer Standard Garbage-Collection Bibliothek wäre dann ein zweiter Schritt, der später folgen könnte und die Portabilität von C++-Programmen steigern würde.

Die Definition neuer Schlüsselwörter oder neuer Symboliken wie in C++/CLI halte ich persönlich für eher untypisch, doch es bleibt abzuwarten, wie stark Microsofts Einfluss im Standardisierungsgremium reicht.

Literatur

[Boehm 2002] Boehm, Hans: Conservative garbage collector / IBM. URL `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`, 2002. – Forschungsbericht

[Goetz 2003] Goetz, Brian: A brief history of garbage collection. In: *IBM: Java theory and practice* (2003), 28 Oktober. – URL `http://www-106.ibm.com/developerworks/java/library/j-jtp10283/`