Garbage Collection in C++

Überarbeitete Fassung der Studienarbeit vom 11.1.93

Wolf Siberski
20.1.95

Inhaltsverzeichnis

1 Einleitung **3**

1.1 Garbage Collection in C++? *3*

1.2 Aufbau der Studienarbeit *4*

1.3 Danksagung *4*

2 Die drei grundlegenden Collector-Algorithmen **5**

2.1 Begriffe *5*

2.2 Der Reference-Counting-Algorithmus *5*

2.3 Der Mark-and-Sweep-Algorithmus *6*

2.4 Der Copy-Algorithmus *9*

2.5 Hybride Algorithmen *10*

3 Variationen der grundlegenden Algorithmen **11**

3.1 Non-Stop-Versionen *11*

3.2 Zusammenarbeit mit virtuellem Speicher *11*

3.3 Generationelle Garbage Collection *12*

3.4 Opportunistische Garbage Collection *12*

3.5 Recycling *13*

4 Garbage Collection in C++ - Probleme und Lösungen **14**

4.1 Voraussetzungen für Garbage Collection in C++ *14*

4.1.1 Die Root 14

4.1.2 Die Zuordnung von Objekten zu Adressen 14

4.1.3 Pointer in Objekten 14

4.1.4 Arrays in C++ 14

4.1.5 Der Destruktoraufruf 15

4.2 Lösungstechniken *15*

4.2.1 Konservative Garbage Collection 15

4.2.2 Smart Pointer 16

4.2.3 Manuell erzeugter Code 17

4.2.4 Compiler-/Sprachmodifikationen 17

5 Realisierte Garbage Collectoren für C++ **18**

5.1 Der konservative Collector von Boehm *18*

5.1.1 Die Allocation 18

5.1.2 Die Collection 18

5.1.3 Vorteile 19

5.1.4 Nachteile 19

5.1.5 Der Collector als Debugging-Tool 20

5.1.6 Die generationelle Version 20

5.1.7 Die (fast) parallele Version 20
5.1.8 Blacklisting 20
5.2 Der Mostly-Copying-Collector von Bartlett *21*
5.2.1 Die Allocation 21
5.2.2 Die Collection 21
5.2.3 Vorteile 24
5.2.4 Nachteile 24
5.2.5 Die generationelle Version 24
5.3 Der Reference-Counting-Collector von Detlefs *25*
5.3.1 Die Allocation 25
5.3.2 Die Collection 26
5.3.3 Vorteile 26
5.3.4 Nachteile 26
5.4 Das Proposal von Ellis/Detlefs *26*
5.4.1 Die Spracherweiterung 27
5.4.2 Die Collector-Algorithmen 27
5.4.3 Der Destruktoraufruf 27
5.4.4 Weak Pointer 27
5.4.5 Anforderungen an die Codeerzeugung 27
5.5 Der Mark-Sweep-Copy-Collector *27*
5.5.1 Die Schnittstelle 27
5.5.2 Die Allocation 28
5.5.3 Die Collection 28
5.5.4 Vorteile 31
5.5.5 Nachteile 32
5.5.6 Ausblick 32
5.6 Vergleichende Untersuchungen *32*
ANHANG **33**
A Glossar **33**
B Bibliographie **35**
C Legende **41**

1 Einleitung

1.1 Garbage Collection in C++?

Garbage Collection wird üblicherweise mit Sprachen wie LISP oder PROLOG in Verbindung gebracht. In imperativen Programmiersprachen ist sie dagegen wenig verbreitet. Das liegt daran, daß in solchen Sprachen bisher immer ablauforientiert programmiert wurde. Dadurch war es meist ohne Schwierigkeiten möglich, festzustellen, zu welchem Zeitpunkt ein Speicherblock nicht mehr gebraucht wird und freigegeben werden kann.

Moderne objekt-orientierte Systeme werden dagegen nicht mehr ablauforientiert programmiert. Statt dem Benutzer einen bestimmten Ablauf vorzuschreiben, bieten sie ihm vielfältige Möglichkeiten an. Durch Auswahl aus diesen Möglichkeiten bestimmt der Benutzer den Ablauf. Dadurch entsteht oft die gleiche Schwierigkeit hinsichtlich der Speicherverwaltung wie in LISP oder PROLOG: Während des Programmierens läßt sich noch nicht vorhersagen, an welcher Stelle des Programmablaufs ein dynamisch angeforderter Speicherblock freigegeben werden darf.

Diese Schwierigkeit soll am Beispiel eines Bürosystems illustriert werden. In diesem Bürosystem gibt es zwei Arten von Objekten, Materialobjekte und Werkzeugobjekte. Die Materialobjekte entsprechen den in einem nicht automasierten Büro verwendeten Materialien: Es gibt z.B. Rechnungen, Verträge, Kundenadressen und Kundenakten. Die Werkzeugobjekte dienen dazu, mit den Materialien umzugehen, d.h. sie anzuschauen und zu bearbeiten: Es gibt z.B. ein Werkzeug „Adressenkartei“ zur Verwaltung der Kundenadressen, einen „Formulareditor“, um Rechnungen und Verträge auszufüllen, und einen „Kundenaktenverwalter“ zum Bearbeiten von Kundenakten.

Folgendes Szenario ist typisch für ein solches System: Das Werkzeug „Adressenkartei“ erzeugt ein Materialobjekt „Kundenadresse“ und gibt einen Zeiger auf dieses Objekt an die Werkzeuge „Formulareditor“ und „Kundenaktenverwalter“ weiter. Welches Werkzeug soll nun das erzeugte Objekt freigeben, wenn es nicht mehr gebraucht wird? Die naheliegendste Lösung scheint es zu sein, daß die Adressenkartei, die das Objekt erzeugt hat, auch für seine Freigabe verantwortlich sein soll. Aber woher soll die Adressenkartei wissen, ob der Formulareditor das Adressenobjekt noch braucht oder nicht? Denkbar wäre es, daß die Adressenkartei eine Nachfragefunktion des Formulareditors aufrufen könnnte, die Auskunft darüber gibt, ob das Adressenobjekt noch gebraucht wird. Diese Lösung hat jedoch (mindestens) zwei Nachteile:

- Die Unabhängigkeit der Werkzeuge voneinander, die dadurch gewährleistet ist, daß kein Werkzeug etwas über die Interna anderer Werkzeuge weiß, müßte aufgegeben werden.
- Die entstehenden Nachfragefunktionen wären sehr komplex und damit fehleranfällig, weil sehr viele Möglichkeiten berücksichtigt werden müßten (z.B. könnte der Formulareditor das Adressenobjekt inzwischen an ein viertes und fünftes Werkzeug weitergegeben haben).

In solch einem Fall ist die ideale Lösung ein Garbage Collector. Er bietet folgende Vorteile:

- Die Werkzeuge bleiben voneinander unabhängig.
- Es treten keine Fehler mehr durch vorzeitig freigegebene Objekte auf.
- Es treten keine Storage Leaks mehr auf (damit sind Objekte gemeint, die nicht freigegeben werden, obwohl sie schon lange nicht mehr benutzt werden).
- Der Implementierungsaufwand für die manuelle Freigabe der Objekte entfällt.
- Da der Garbage Collector eine gewissermaßen physikalische Sicht auf die Objekte hat, während die Werkzeuge eine logischen Sicht auf die Objekte haben, ist auch die Unabhängigkeit des Collectors von den Werkzeugen gewährleistet, d.h. eine Änderung des Programms (z.B. Hinzufügen neuer Werkzeuge) wirkt sich nicht auf den Collector aus.

Diese Vorteile sind der Grund dafür, daß in letzter Zeit auch für Sprachen wie C++ Garbage Collectoren entwickelt werden.

1.2 Aufbau der Studienarbeit

Es gibt drei Garbage-Collector-Algorithmen, die die Grundlage für alle Weiterentwicklungen bieten, nämlich Mark-and-Sweep-Collection, Copy-Collection und Reference-Counting-Collection. Diese drei Algorithmen werden in Kapitel 2 dargestellt. Den ersten Abschnitt dieses Kapitels, den Abschnitt *Begriffe,* sollte jeder lesen. Die dort vorgestellten Begriffe werden in der Studienarbeit ständig verwendet. Den Rest des Kapitels kann derjenige Leser, der diese Algorithmen schon kennt, überschlagen.

Die drei Grundalgorithmen sind in verschiedene Richtungen weiterentwickelt worden. Diese Entwicklungen werden in Kapitel 3 dargestellt.

Die Implementierung eines Collectors in C++ bietet besondere Probleme. Diese Probleme und mögliche Lösungen werden in Kapitel 4 beschrieben.

In Kapitel 5 werden einige Collectoren vorgestellt, die die im Kap. 4 vorgestellten Ansätze verwirklicht haben. Einer der beschriebenen Collectoren ist der im Rahmen dieser Studienarbeit entstandene Mark-Sweep-Copy-Collector.

1.3 Danksagung

An erster Stelle möchte ich mich besonders bei Heinz Züllighoven bedanken, ohne dessen intensive Betreuung und ständige Gesprächsbereitschaft diese Studienarbeit nie zustandegekommen wäre.

Für die inhaltliche und technische Unterstützung bedanke ich mich bei allen Mitarbeitern des Arbeitsbereichs Softwaretechnik. Insbesondere geholfen haben mir Ernst Maracke und Guido Gryczan, der mich in die Geheimnisse des Internet einweihte.

Wertvolle Anregungen und Kritik verdanke ich Dirk Riehle und Kurt Stege, die diese Studienarbeit in verschiedenen Fassungen gelesen haben.

Last but not least gilt mein Dank den Mitarbeitern des Projekts GEBOS Passiv der RWG GmbH, die es mir ermöglicht haben, meine Konzepte praktisch umzusetzen.

2 Die drei grundlegenden Collector-Algorithmen

Dieses und das nächste Kapitel basieren im Wesentlichen auf den Überblicksartikeln zu Garbage Collection von Cohen [Cohen81] und Wilson [Wilson94].

2.1 Begriffe

Die Begriffe **Objekt**, **Pointer** und **Adresse** werden hier folgendermaßen verwendet:

Objekt Ein Objekt ist ein zusammenhängender Speicherblock, in dem e Variable gespeichert ist. Diese Variable kann auch im OOP-Sinne Objekt sein, muß es aber nicht. Diese Bedeutung lehnt sich an Verwendung des Wortes Objekt in „The C++ Programming Lang ge“ [Stroustrup91] an.

Pointer Ein Pointer ist eine Referenz auf ein Objekt; das kann die Sp cheradresse des Objekts sein, muß es aber nicht. Entscheidend ist, über den Pointer ein Zugriff auf das Objekt möglich ist.

Adresse Mit Adresse eines Objekts ist immer die Maschinenadresse des Sp cherblocks gemeint, in dem das Objekt gespeichert ist.

Diese Verwendung der Begriffe Objekt und Pointer ist zwar etwas ungewöhnlich, erleichtert aber die Formulierung der Garbage-Collector-Algorithmen sehr.
Zusätzlich haben sich die folgenden Begriffe eingebürgert:

Collector Als Collector wird derjenige Prozeß/Programmteil bezeichnet, der die dynamische Speicherverwaltung zuständig ist (Allocation u Collection).

Mutator Als Mutator wird derjenige Prozeß oder Programmteil bezeichnet, die dynamische Speicherverwaltung benutzt.

Allocation Die Bearbeitung einer Speicheranforderung des Mutators durch Collector heißt Allocation.

Collection Die Phase des Findens von nicht mehr benutzten Objekten und erneuten Bereitstellung ihres Speichers wird Collection genannt.

Root Die Root ist die Menge aller Pointer, von denen der Collector ausge um alle benutzten Objekte zu finden.

benutztes Objekt Ein Objekt heißt dann benutzt, wenn es innerhalb des Mutators r einen Pointer auf dieses Objekt gibt.

gefundenes Objekt Ein Objekt, von dem der Collector bereits erkannt hat, daß es n benutzt wird, wird gefundenes Objekt genannt.

2.2 Der Reference-Counting-Algorithmus

Der Reference-Counting-Algorithmus funktioniert folgendermaßen: Jedem Objekt wird ein Zähler zugeordnet, der die Anzahl der Pointer enthält, die auf dieses Objekt zeigen. Bei jeder Veränderung des Inhalts eines Pointers (z.B. durch Zuweisung) werden die Zähler entsprechend modifiziert; der Zähler des Objekts, auf das der Pointer vorher zeigte, wird dekrementiert, und der Zähler des Objekts, auf das der Pointer jetzt zeigt, wird inkrementiert.
Durch diese Vorgehensweise sind zwei Bedingungen immer erfüllt:

- Alle noch benutzten Objekte haben einen Zähler ungleich 0.
- Alle Objekte mit einem Zähler gleich 0 werden nicht mehr benutzt.

Leider gibt es auch unbenutzte Objekte mit einem Zähler ungleich 0, da Objekte in einer unbenutzten zyklischen Struktur auch einen Zähler ungleich 0 haben.

Dieser Algorithmus hat zwei Vorteile: Erstens wird zum frühestmöglichen Zeitpunkt erkannt, daß ein Objekt nicht mehr benutzt wird (Ausnahme: zyklische Strukturen!). Zweitens muß der Collector den Mutator nie für längere Zeit unterbrechen, da die gesamte Arbeit bei Pointerzuweisungen getan wird - pro Pointerzuweisung kurz und in konstanter Zeit!
Diese Vorteile werden allerdings teuer erkauft. Der wesentliche Nachteil ist, daß der Collector immer Zeit kostet, auch dann, wenn er gar nicht gebraucht wird. Dies ist in folgenden Situationen der Fall:

1. Der Mutator braucht so wenige Objekte, daß ständig genügend Speicher vorhanden ist.
2. Der Mutator benutzt während der gesamten Laufzeit (fast) alle Objekte, die er angefordert hat.
3. Der Mutator gibt einen Großteil der nicht mehr benutzten Objekte selbst an die Speicherverwaltung zurück.

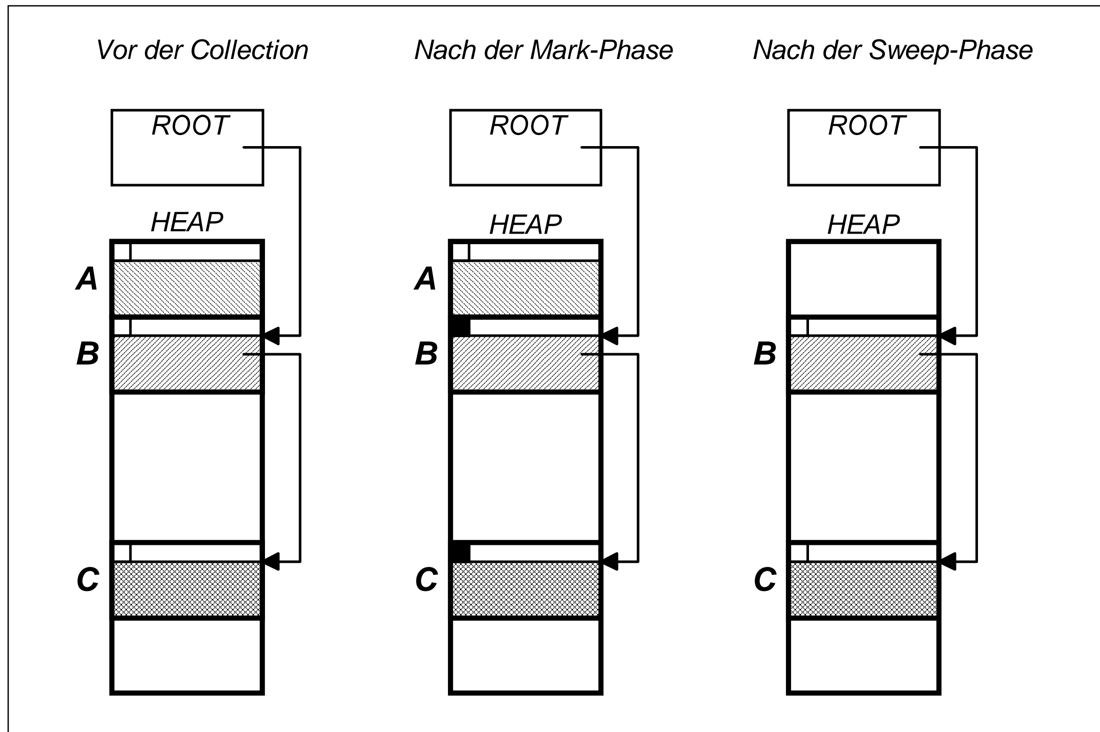
Der Algorithmus wurde für funktionale Programmiersprachen wie LISP entwickelt, in denen die obengenannten Situationen fast nie (1, 2) oder nie (3) vorkommen. Dagegen treten in imperativen objektorientierten Programmiersprachen wie C++[1] solche Situationen häufig auf, besonders, weil der Speicherplatz für lokale Variablen nicht von der dynamischen Speicherverwaltung angefordert, sondern vom Stack genommen wird (dadurch wird er automatisch beim Verlassen des Geltungsbereichs der lokalen Variablen freigegeben) [Hölzle91].
Weitere Nachteile sind der erhöhte Speicherbedarf für die benötigten Zähler und die Unfähigkeit des Algorithmus, zyklisch verkettete unbenutzte Objekte in die Freimenge aufzunehmen.
Aufgrund dieser Nachteile wird der Reference-Counting-Algorithmus hier nicht weiter betrachtet. Weitergehende Informationen findet man in [Cohen81], Abschnitt 4.

2.3 Der Mark-and-Sweep-Algorithmus

Der Mark-and-Sweep-Algorithmus besteht aus zwei Phasen. In der ersten Phase, der Mark-Phase, werden alle benutzten Objekte markiert. In der zweiten Phase, der Sweep-Phase werden alle nicht markierten Objekte freigegeben.
Zur Durchführung der Mark-Phase besitzt jedes Objekt ein Mark-Bit. Ihre Implementierung erfolgt meistens mit Hilfe eines Pointer-Stacks. Zunächst werden alle Root-Pointer auf den Stack gepusht. Dann wird folgende Aktion durchgeführt: Der oberste Pointer wird vom Stack gepopt; falls das Objekt, auf das er zeigt, noch nicht markiert ist, wird es markiert, und die in ihm enthaltenen Pointer auf den Stack gepusht. Falls es schon markiert ist, geschieht nichts. Diese Aktion wird solange wiederholt, bis der Stack leer ist. Dann sind alle benutzten Objekte markiert.
Es sind auch Mark-Algorithmen entwickelt worden, für die nur ein Stack fester Größe oder sogar gar kein Stack benötigt wird; die Information, die üblicherweise auf dem Stack abgelegt wird, wird bei diesen Algorithmen in den Objekten gespeichert. Informationen dazu gibt [Cohen81], Abschnitt 1.1.
Die Sweep-Phase besteht darin, jedes Objekt im Heap zu untersuchen und es, falls es nicht markiert ist, freizugeben. Dazu wird entweder die Adresse des Objekts in eine Freiliste aufgenommen oder das Objekt wird in eine verkettete Liste aller freien Objekte eingefügt.
Eine Beispielcollection zeigt die Abbildung auf der folgenden Seite[2]. Zu Beginn der Collection existieren innerhalb des Programms nur noch Pointer auf die Objekte B und C, d.h. das Objekt A wird nicht mehr benutzt (linkes Bild). Während der Mark-Phase findet der Collector zuerst beim Durchsuchen der Root dein Pointer auf B und markiert daher B (schwarzes Quadrat in der linken oberen Ecke des Objekts). Als nächstes untersucht er B und findet das Objekt C, das er ebenfalls markiert. Da C keine Pointer enthält, ist danach die Mark-Phase beendet (mittleres Bild). Während der Sweep-Phase prüft der Collector der Reihe nach die Mark-Bits der Objekte A, B und C. Da das Mark-Bit von A nicht gesetzt ist, wird A freigegeben. An den gesetzten Mark-Bits von B und C erkennt der

[1] In C++ wird der Zeitbedarf für diesen Algorithmus dadurch noch vergrößert, daß aus technischen Gründen auch der lesende Zugriff auf Pointer länger dauert (dies ließe sich nur durch eine Änderung der Sprachdefinition vermeiden)
[2] Zur Erklärung der verwendeten Zeichen siehe Seite 35.

Collector, daß diese Objekte noch benutzt werden. Er beschränkt sich daher darauf, ihre Mark-Bits zurückzusetzen (rechtes Bild).



Der Mark-and-Sweep-Algorithmus läßt sich in C++-Pseudocode folgendermaßen formulieren:

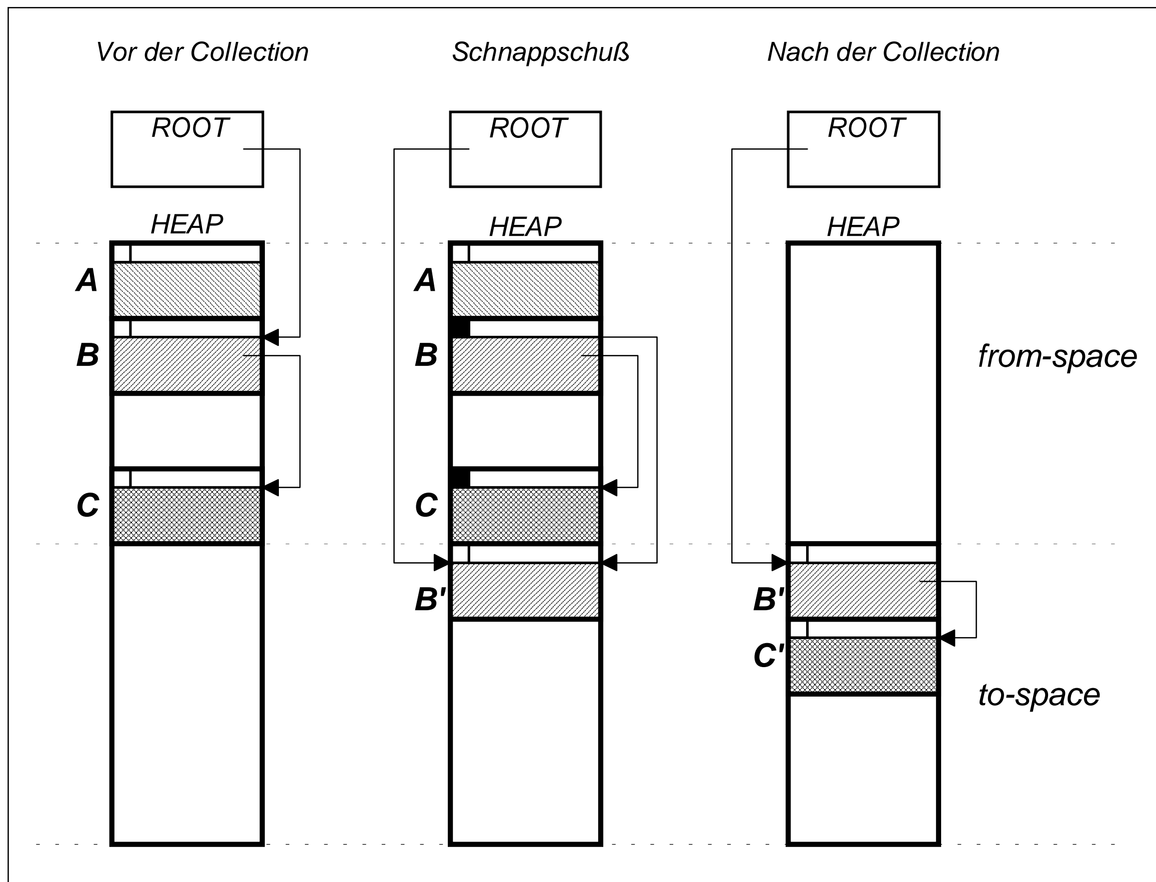
```
static CStack Stack;

void Collect() {
     PushRoots();
     Mark();
     Sweep();
}

void PushRoots() {
     Obj *p;

     for( each root-pointer p )
          Stack.Push( p );
}

void Mark() {
     Obj *p, *q;

     while ( ! Stack.IsEmpty() ) {
          p = Stack.Pop();
          if ( ! p->IsMarked() ) {
               p->Mark();
               for ( each pointer q in p )
                    Stack.Push( q );
          }
     }
}

void Sweep() {
     Obj* o;

     for ( each object o ) {
          if ( ! o->IsMarked() )
               Free( o );
     }
```

8

}

2.4 Der Copy-Algorithmus

Der Copy-Collector kopiert die benutzten Objekte während der Collection in einen freien Teil des Speichers. Da im schlechtesten Fall alle Objekte noch benutzt werden, muß also immer die Hälfte des Speichers für den Collector reserviert bleiben (bzw. in Systemen mit virtuellem Speicher die Hälfte des Adressraums). Aus diesem Grund wird der Speicher (bzw. der Adressraum) in zwei gleich große Teile geteilt. Vor der Collection dürfen Objekte nur in einem dieser Teile alloziert werden. Dieser Teil heißt **from-space**. In den anderen Teil, den sogenannten **to-space** werden die Objekte während der Collection kopiert.
Da alle Pointer während der Collection auf die neuen Objektadressen gesetzt werden müssen, wird die neue Adresse jedes Objekts an seiner alten Adresse abgelegt und ein Mark-Bit gesetzt, um anzuzeigen, daß es schon kopiert worden ist. Immer dann, wenn der Collector beim Untersuchen eines gefundenen Objekts auf einen Pointer stößt, der in den from-space zeigt, wird er auf die neue Adresse umgesetzt.
Da sich nach der Collection im from-space keine erreichbaren Objekte mehr befinden, kann der gesamte from-space freigegeben werden. Er dient üblicherweise im nächsten Durchlauf dem Collector als neuer to-space.



Obige Abb. zeigt ein Beispiel einer solchen Collection. Der Collector durchsucht die Root nach Pointern in den from-space. Als erstes findet er den Zeiger auf B. Er legt daraufhin eine Kopie von B im to-space an (B')und markiert B als kopiert. Anschließend wird der Zeiger in der Root von B auf B' umgesetzt. Das mittlere Bild zeigt die entstandene Lage. Auf die gleiche Art wird nun C kopiert und der Zeiger in B' von C auf C' umgesetzt. Nun sind alle benutzten Objekte in den to-space kopiert und der from-space kann komplett freigegeben werden.
Hier der Copy-Algorithmus in C++-Pseudocode:

```
static CStack Stack;

void Collect() {
```

```
        PushRootAdr();
        Copy();
    }

    void PushRootAdr() {
        Obj *p;

        for( each root-pointer p )
            Stack.Push( &p );
    }

    void Copy() {
        Obj **p, *q;

        while ( ! Stack.IsEmpty() ) {
            p = Stack.Pop();
            if ( ! (*p)->IsCopied() ) {
                CopyObj( p );
                *p->MarkAsCopied();
                for ( each pointer q in p )
                    Stack.Push( &q );
            }
            else {
                *p = GetNewAdr( p );
            }
        }
    }
```

2.5 Hybride Algorithmen

Es sind verschiedene Collectoren entwickelt worden, die die grundlegenden Algorithmen kombinieren. Reference-Counting-Collectoren haben üblicherweise einen zusätzlichen Algorithmus, um zyklische Strukturen freigeben zu können. Copy-Collectoren für C++, wie der von Bartlett [Bartlett90a], dürfen manche Objekte nicht kopieren (näheres s. 5.2) und müssen daher ebenfalls hybride Techniken verwenden.

Als Beispiel für hybride Algorithmen sei hier der Collector von Lang/Dupont vorgestellt, eine Kombination eines Mark-and-Sweep- mit einem Copy-Collector [LanDup87]. Lang/Dupont versuchen damit, die Vorteile beider Arten miteinander zu verbinden.

Der Collector alloziert zu Beginn vom Betriebssystem den Speicher für den kompletten Heap. Er wird in n+1 Seiten aufgeteilt, die von 0 bis n durchnummeriert werden. Eine Seite wird für die nächste Garbage Collection als to-Space reserviert, die anderen n werden in eine Free-List eingetragen. Diese Free-List ist immer nach der Größe der freien Speicherblöcke sortiert. Wenn der Mutator Speicher für ein Objekt anfordert, wird nach der best-fit-Strategie ein passender Speicherblock gesucht. Ist er größer als nötig, wird er in einen passenden Block und einen Restblock geteilt und der Restblock in die Free-List einsortiert.

Die Collection läuft folgendermaßen ab:

Die Nummer derjenigen Seite, die als to-Space reserviert wurde, wird mit i bezeichnet. Zu Beginn werden die Mark-Bits aller Objekte gelöscht. Von der Root ausgehend, werden die erreichbaren Objekte markiert. Wenn ein Objekt gefunden wird, das auf Seite i+1 liegt, wird es außerdem auf die Seite i kopiert und an seiner alten Adresse ein Pointer auf die neue Adresse abgelegt. Wenn es schon kopiert worden ist, wird nur der Pointer auf die neue Adresse gesetzt.

Nach Abschluß dieser Mark-Copy-Phase wird für alle außer den Seiten i und i+1 eine Sweep-Phase durchgeführt, in der alle unmarkierten Objekte in die Free-List eingetragen werden. Hintereinanderliegende freie Objekte werden dabei zu einem Objekt vereinigt.

Am Ende der Collection sind alle erreichbaren Objekte aus Seite i+1 in Seite i kopiert und die Seite i+1 ist leer. Sie wird als to-Space für die nächste Collection reserviert.

Der Vorteil des Collectors besteht darin, daß die Fragmentierung des Speichers beseitigt wird (wenn auch langsamer als beim Copy-Collector), daß er aber nicht einen Speicher von der doppelten Größe des Heaps benötigt, sondern nur einen 1/n größeren.

3 Variationen der grundlegenden Algorithmen

Die bisher vorgestellten Algorithmen sind in verschiedene Richtungen weiterentwickelt worden:

3.1 Non-Stop-Versionen

Konventionelle Collectoren stoppen den Mutator und führen dann die gesamte Collection durch. Erst danach lassen sie den Mutator weiterlaufen. Dieses Verhalten ist in folgenden Fällen oft nicht akzeptabel:

1. Die zugrundeliegende Maschine ist ein Mehrprozessor-System; dann möchte man Mutator und Collector als Prozesse auf verschiedenen Prozessoren implementieren, um die Leistungsfähigkeit des Systems nutzen zu können.
2. Der Mutator ist eine Echtzeit-Anwendung. Dann darf der Collector den Mutator nur für eine kurze und nach oben beschränkte Zeit unterbrechen, damit der Mutator seine Zeitrestriktionen erfüllen kann.
3. Der Mutator ist eine interaktive Anwendung. Dann muß der Collector bei einer Benutzereingabe unterbrechbar sein; sonst müßte der Benutzer eventuell unzumutbar lange warten.

Für alle drei Fälle sind Collectoren entwickelt worden:

1. **Parallele Collectoren**: Diese Collectoren sind so geschrieben, daß möglichst wenig explizite Synchronisation mit dem Mutator nötig ist, damit die beiden Prozesse nicht voneinander aufgehalten werden.
2. **Echtzeit-Collectoren**: Bei diesen Collectoren wird der Zeitaufwand möglichst gleichmäßig über die Laufzeit verteilt, und zwar so, daß der Mutator nur für eine bestimmte Maximalzeit unterbrochen wird.
3. **Inkrementelle Collectoren**: Hier sind Mutator und Collector zwar verschiedene Prozesse, die aber auf einem Prozessor laufen. Inkrementelle Collectoren sind so geschrieben, daß sie auch teilweise Collectionen durchführen können. Oft sind sie so konzipiert, daß jede Collection schnell abgebrochen werden kann.
   Üblicherweise wird ein solcher Collector zu geeigneten Zeitpunkten vom Mutator als quasi-nebenläufiger Prozeß gestartet und, z.B. bei einer Benutzereingabe, auch wieder abgebrochen.

Die Hauptschwierigkeit bei der Konzeption eines non-stop-Collectors besteht darin, einerseits die Konsistenz der Objekte trotz gemischten (quasi gleichzeitigen) Zugriffs des Mutators und des Collectors zu sichern und andererseits beide Prozesse möglichst unabhängig voneinander arbeiten zu lassen.
Die Algorithmen für die Synchronisation in herkömmlichen Speicherverwaltungen hier zu beschreiben würde den Rahmen dieser Arbeit überschreiten. Ausführliche Informationen sowie Verweise auf weiterführende Literatur finden sich in [Cohen81] und [Wilson94].

3.2 Zusammenarbeit mit virtuellem Speicher

In einem System mit virtuellem Speicher verschiebt sich die Aufgabe eines Collectors. In konventionellen System ging es darum, überhaupt genügend Speicher für den Mutator bereitzustellen; in Systemen mit virtuellem Speicher bereitet dies üblicherweise keine Probleme, da der Adressraum genügend groß ist. Allerdings nimmt die Zeit, die die Speicherverwaltung benötigt, mit der Menge der Speicher-Seiten, die auf Massenspeicher ausgelagert und bei Zugriffen wieder geladen werden müssen, zu. Dementsprechend nimmt die Zeit, die dem Mutator zur Verfügung steht, ab.
Daher ist es eine Aufgabe des Collectors, die Anzahl der Seiten, die vom Mutator benutzt werden, möglichst klein zu halten. Außerdem sollte der Collector die Objekte so auf die Seiten verteilen, daß möglichst selten Seiten ein- und ausgelagert werden müssen.

Systeme mit virtueller Speicherverwaltung bieten üblicherweise auch die Möglichkeit, Seiten gegen Schreibzugriffe zu schützen und festzustellen, ob seit einem bestimmten Zeitpunkt auf eine Seite geschrieben worden ist (**dirty bit**). Beide Features lassen sich dazu benutzen, den Collector zum größten Teil parallel zum Mutator arbeiten zu lassen, ohne daß explizite Synchronisation mit ihm erforderlich ist. Es gibt zwei Ansätze in dieser Richtung:
Der erste Ansatz ist eine Weiterentwicklung des Mark-and-Sweep-Algorithmus und nutzt das Dirty-Bit-Feature [DemWeiHay+90]: Zunächst läßt man den Collector wie immer eine Collection durchführen, während der Mutator weiterläuft. Dann stoppt man den Mutator, um die Seiten nachzubearbeiten, die inzwischen von ihm verändert worden sind. Die dadurch entstehende Pause ist fast immer erheblich geringer als im Falle eines Collectors, der den Mutator während der gesamten Collection stoppt.
Der zweite Ansatz entstand aus dem Copy-Algorithmus [Appel89]: Hier wird das Schreibschutz-Feature genutzt. Der Collector wird als paralleler Prozeß gestartet. Der Mutator darf nur auf Adressen im to-space zugreifen. Wenn er auf eine Adresse im from-space zugreift, wird eine Exception ausgelöst, die der Collector folgendermaßen bearbeitet: Er kopiert als nächstes die Objekte der Seite, auf die der Mutator zugreifen wollte und ersetzt den Pointer, der die Exception auslöste, durch den neuen Pointer auf dasselbe Objekt im to-space. Damit ist die Ausnahmebehandlung beendet und Mutator wie Collector können ihre normale Arbeit fortsetzen. Wie im ersten Ansatz werden die Pausen des Mutators erheblich verkürzt.

3.3 Generationelle Garbage Collection

Der Ansatz der generationellen Garbage Collection [Ungar84] beruht auf zwei Beobachtungen:

- Objekte, die schon längere Zeit in Benutzung gewesen sind, haben eine hohe weitere „Lebenserwartung“, während Objekte, die gerade erst alloziert worden sind, eine geringe „Lebenserwartung“ haben.
- Meistens zeigen die Pointer von jüngeren Objekten auf ältere Objekte und nicht umgekehrt.

Diese Beobachtungen werden folgendermaßen ausgenutzt: Die Objekte werden in mehrere Generationen eingeteilt, die vom Collector unterschiedlich behandelt werden; der Einfachheit halber nehme ich im folgenden zwei Generationen an. Ein frisch alloziertes Objekt wird Element der ersten Generation. Sobald ein Objekt eine festgelegte Anzahl (üblicherweise zwischen 1 und 4) an Collections „überlebt“ hat, d.h., benutztes Objekt geblieben ist, wird sie Element der zweiten Generation, der **stabilen Menge**. Falls von diesem Objekt Pointer auf Objekte der ersten Generation zeigen, werden diese Pointer in einer speziellen Liste eingetragen. Diese Liste muß durch explizite Kooperation des Mutators oder mit Hilfe eines Dirty-Bit immer auf dem neuesten Stand gehalten werden.
Die stabile Menge wird von nun an für die Collection ignoriert. Dafür wird die Liste mit den Pointern auf Objekte der ersten Generation zur Root hinzugenommen. Nur, wenn durch Untersuchen der ersten Generation nicht mehr genügend freie Objekte gewonnen werden können, bezieht der Collector die stabile Menge in die Collection mit ein. Eine solche Collection wird **volle Collection** genannt.
Dieses Vorgehen hat den Vorteil, daß weniger Objekte durchsucht werden müssen, was zu einer Verringerung der Durchschnittsdauer einer Collection führt. Die Maximaldauer verkürzt sich jedoch nicht, da ab und zu (wenn auch selten) eine volle Collection notwendig wird.

3.4 Opportunistische Garbage Collection

Die Idee der opportunistischen Garbage Collection besteht darin, den Zeitpunkt, zu dem eine Collection durchgeführt wird, geschickt zu wählen [Wilson88]. Dies kann in zweierlei Hinsicht geschehen: Erstens kann (in interaktiven Systemen) der Zeitpunkt so gewählt werden, daß die Collection keine für den Benutzer unzumutbare Pause bewirkt. Zweitens kann der Zeitpunkt so gewählt werden, daß der Collector besonders viele freie Objekte findet. Collectionen sind z.B. zu folgenden Zeitpunkten besonders günstig:

- Am Ende längerer Rechenphasen; denn wenn der Benutzer schon 30 Sekunden auf das Rechenergebnis wartet, ist es zumutbar, ihn noch eine Sekunde warten zu lassen.

- Dann, wenn der Benutzer schon längere Zeit keine Eingabe gemacht hat; denn dann ist es wahrscheinlich, daß er auch in der nächsten Sekunde keine Eingabe machen wird.
- Dann, wenn der Stack gerade wenig gefüllt ist; denn dann ist die Menge der benutzten Objekte im Durchschnitt geringer als bei stark gefülltem Stack.

Der besondere Vorteil dieses Ansatzes ist, daß er leicht zu implementieren ist und zusammen mit jedem anderen Ansatz verwendet werden kann.

3.5 Recycling

In jüngster Zeit ist versucht worden, aus Garbage wieder sinnvolle Daten zu gewinnen. Dazu werden neuartige Datenstrukturen verwendet, wie z.B. die sogenannte „arab hen“, die Garbage in „arab egg“ umwandelt. Praktische, über Experimente hinausgehende Erfahrungen sind bislang noch nicht gesammelt worden. Weitere Informationen enthält [Creak91] :-).

4 Garbage Collection in C++ - Probleme und Lösungen

4.1 Voraussetzungen für Garbage Collection in C++

Wie schon oben erwähnt, gehen alle drei GC-Algorithmen davon aus, daß diejenigen Objekte als benutzt angesehen werden, die über Pointer erreichbar sind. Um diese Objekte finden zu können, müssen folgende Voraussetzungen erfüllt sein:

- Die Root muß festgelegt sein.
- Der Collector muß zu einem gegebenen Pointer bzw. einer gegebenen Adresse das entsprechende Objekt finden können.
- Der Collector muß zu einem gegebenen Objekt alle im Objekt enthaltenen Pointer finden können

Außerdem gibt es noch folgende erwünschte, aber nicht notwendige Eigenschaften:

- Der Collector soll mit C++-Arrays umgehen können
- Der Collector soll zu jedem freigegebenen Objekt seinen Destruktor aufrufen.

Diese Voraussetzungen werden im folgenden erläutert.

4.1.1 Die Root

Wie die Root festgelegt werden muß, ist bei üblichen C++-Programmen klar: sie besteht aus den Pointern in statischen und automatischen(=Stack-) Variablen. Auf der Maschinencode-Ebene sind das die Adressen im Daten- und Stacksegment und in den Prozessorregistern.
Es gibt aber auch Möglichkeiten, die Root zu bestimmen, ohne die C++-Quelltextebene zu verlassen.

4.1.2 Die Zuordnung von Objekten zu Adressen

Diese Voraussetzung ist auf Quelltextebene völlig unproblematisch, wenn alle Klassen (indirekt) von einer Wurzelklasse abgeleitet sind; dies ist sowieso nötig, da jede Klasse bestimmte Methoden bereitstellen muß, um GC-tauglich zu werden.
Bei den meisten Collectoren für C++ wird jedoch die Root auf Maschinen-Ebene bestimmt. Dabei genügt es nicht, den Anfang jedes Objekts zu kennen, weil es dort virtuelle Basisklassen gibt. Objekte mit virtuellen Basisklassen haben nämlich eine unangenehme Eigenschaft: Es kann passieren, daß kein einziger Pointer auf ihren Anfang zeigt, dafür aber manche Pointer, nämlich Pointer vom Typ einer virtuellen Basisklasse, in ihr Inneres. Ähnliche Probleme tauchen auch im Zusammenhang mit C++-Arrays auf.

4.1.3 Pointer in Objekten

Der Collector muß in der Lage sein, den Referenzen innerhalb von Objekten folgen zu können, um die benutzten Objekten markieren bzw. kopieren zu können. Dazu gibt es verschiedene Methoden, die unter den einzelnen Techniken beschrieben werden.

4.1.4 Arrays in C++

Ein besonderes Problem bilden in C++ die Arrays. Das liegt daran, daß einerseits ein C++-Array kein eigenständiges Objekt ist, sondern eine Folge mehrerer unabhängiger Objekte, andererseits das Array durch einen `new`-Aufruf erzeugt wird und daher vom Collector wie ein Objekt behandelt werden muß.
Die einfachste Lösung für dieses Problem besteht darin, C++-Arrays zu verbieten und durch eine Array-Klasse zu ersetzen; dadurch wird ein Array zu einem Objekt wie jedes andere. Da das Konstrukt "C++-Array" nur ein Anachronismus ist, der aufgrund der C-Kompatibilität in die Sprache gelangt ist, wäre ein Verbot von C++-Objekt-Arrays kein Verlust. Der Vorschlag, sie zu verbieten, ist auch im Proposal von Ellis/Detlefs enthalten [Ell/Det93]

4.1.5 Der Destruktoraufruf

Ein weiteres Problem bietet die Tatsache, daß beim Löschen eines C++-Objekts sein Destruktor aufgerufen werden muß.
Für einen Mark-and-Sweep-Collector ist das auf den ersten Blick nicht schwer: In der Sweep-Phase kann für jedes nicht mehr benutzte Objekt ein delete-Aufruf getätigt werden, sodaß vom zu löschenden Objekt aus gesehen kein Unterschied mehr zur manuellen Speicherverwaltung besteht. In Copy-Collectoren dagegen wird üblicherweise ein großer Speicherblock freigegeben, ohne die einzelnen in ihm enthaltenen Objekte irgendwie zu behandeln.
Von einem Problem sind jedoch beide Collectoren betroffen. Es läßt sich am besten an einem Beispiel deutlich machen. Angenommen, der Destruktor von Objekt A greift auf das Objekt B zu, auf das keine andere Referenz mehr existiert. Wenn der Collector den Destruktor von B zuerst aufruft, kann der Destruktor von A nicht mehr ohne Programmfehler ausgeführt werden. Falls A und B (über andere Objekte) zyklisch verkettet sind, hat der Collector keine Möglichkeit mehr, die korrekte Reihenfolge des Destruktoraufrufs herauszufinden.
Auch für dieses Problem ist eine Lösung, Destruktoren zu verbieten. Im Gegensatz zu den C++-Arrays gibt es aber keinen vollwertigen Ersatz für sie. Denn Destruktoren werden zwar oft benutzt um Speicherplatz freizugeben, was nun nicht mehr nötig ist, aber das ist nicht ihre einzige Verwendung. Häufig werden auch andere Resourcen vom Destruktor freigegeben (File-Handles, Netzwerk-Sokkets, Threads, usw.). Daher kann man auf Destruktoren nicht verzichten.
Eine pragmatische Lösung ist es, wenn der Programmierer zu jeder Klasse angeben muß, daß ihr Destruktor vom Collector aufgerufen werden muß. Die Überprüfung, ob in einem solchen Destruktor auf andere dynamische Objekte zugegriffen wird, könnte entweder der Programmierer selbst oder der Compiler vornehmen. Eine Verwaltung dieser zu destruierenden Objekte kann mit Hilfe sog. Guardians realisiert werden [DybBruEby93].

4.2 Lösungstechniken

Die folgenden Abschnitte beschäftigen sich mit den Techniken, mit denen die obigen Voraussetzungen realisiert werden können.

4.2.1 Konservative Garbage Collection

Die konservative Garbage Collection bietet die Möglichkeit, Pointer im Speicher zu finden, ohne daß dem Collector Informationen über die innere Struktur eine Objekts zugänglich sind. Boehm und Weiser [BoeWei88] fanden für dieses Problem eine verblüffend einfache Lösung. Sie interpretieren alle Daten in dem Speicherbereich, in dem Pointer gesucht werden, als Pointer. Solche Daten, von denen der Collector nicht genau weiß, ob sie Pointer sind, heißen **unsichere Pointer**.
Um die Root zu erhalten, untersuchen sie auf diese Art die Prozessorregister, den Stack und alle statischen Daten. Während der Mark-Phase werden dementsprechend alle in einem gefunden Objekt gespeicherten Daten als Pointer interpretiert.
Dabei kommt es zwar vor, daß Daten, die keine Pointer sind, fehlerhaft als Pointer interpretiert werden, wenn sie dasselbe Bitmuster wie ein Pointer haben. Dadurch werden unter Umständen unbenutzte Objekte nicht als frei erkannt. Die Anzahl solcher Fehlinterpretationen kann jedoch sehr gering gehalten werden, indem man jeden potentiellen Pointer einigen Plausibilitätstests unterwirft, bevor man ihn als Pointer anerkennt.
In sehr ungewöhnlichen Situationen kann die Zahl der Fehlinterpretationen jedoch extrem stark ansteigen, sodaß fast keine freien Objekte erkannt werden [Wentworth90].
Wenn als zulässige Pointer nicht nur Pointer auf den Anfang des Objekts zugelassen werden (und das ist bei Verwendung virtueller Basisklassen erforderlich, s.o.), wird entweder die Anzahl der fälschlich als benutzt markierten Objekte erheblich größer oder die Prüfmethode muß aufwendiger gestaltet werden, was den Zeitbedarf für die Prüfung erhöht.
Boehm hat dieses Problem mit einer neuen Methode (Blacklisting) weitgehend entschärft [Boehm93] (s. 5.1.8).

Dadurch, daß die Objekte komplett durchsucht werden müssen, anstatt gezielt auf die enthaltenen Pointer zuzugreifen, wird der Zeitaufwand relativ hoch. Dafür muß jedoch weder der Programmierer noch der Compiler irgendwelche Informationen für den Collector erzeugen.
Probleme können jedoch hoch optimierende Compiler verursachen, wenn sie den Code so stark optimieren, daß keine Adresse mehr auf ein noch benutztes Objekt zeigt. Ein Beispiel dafür ist folgende Schleife (das Beispiel ist [EllDet93] entnommen):

```
char* a = new char[10];
for( int i=10; i<20; i++ ) {
     a[i-j] = ...
}
```

Der Compiler könnte dafür Code erzeugen, der etwa folgenden C++-Code entspricht:

```
char* a = new char[10];
char* p = a + 10;
char* end = p + 20;
for (; p < end; p++ ) {
     *(p - j) = ...
}
```

Falls der Collector während der for-Schleife sammelt, findet er nur den Pointer p auf dem Stack, der jedoch hinter das Array a zeigt. Daher hält er fälschlicherweise a für unbenutzt. Um dieses Problem zu vermeiden, muß man entweder die Optimierung des Compilers abschalten oder einen Compiler verwenden, dessen Optinierung auf konservative GC Rücksicht nimmt [Boehm91] (s. auch 5.4.5).
Hier der Pseudocode für die konservative Methode.

```
void PushGuessedPointersIn( Obj* p ) {
     long Size = GetObjSize( *p );
     for ( void** q = (void**)p;
            (long) q <= (long)p + Size - PtrSize;
            q = (void**)((long)q + PtrAlignment;
     )
           if ( PointsToAnObj( (Obj*)(*q) ) )
                Stack.Push( (Obj*)(*q) );
}
```

Eine Beschreibung der Details findet sich im Abschnitt \ref{ Boehms Collector }.
Auch die Collectoren von Bartlett und Detlefs verwenden die konservative Methode, um die Root zu bestimmen.

4.2.2 Smart Pointer

Smart-Pointer sind Objekte, die die Standard-C++-Pointer ersetzen und zusätzliche Funktionalität ermöglichen. Da sie die beiden Pointer-Operatoren '*' und '->' überladen, kann man mit ihnen wie mit Standard-Pointern auf die referenzierten Objekte zugreifen. Zusätzliche Funktionalitäten (wie z.B. das Zählen von Referenzen oder das An- und Abmelden bei einem Collector) können im Konstruktor/Destruktor und in den überladenen Operatoren realisiert werden. Damit ist es möglich, einen Collector auf C++-Quelltextebene zu implementieren.
Leider haben Smart-Pointer gravierende Nachteile:

- Das Casten nicht möglich bzw. sehr erschwert. Insbesondere das automatische Casting auf Basisklassen funktioniert z.T. nicht mehr.
- Da keine Standard-Pointer verwendet werden dürfen, lassen sich vorhandene Bibliotheken nicht ohne größere Modifikationen verwenden
- Das Laufzeitverhalten verschlechtert sich stark (etwa um den Faktor 1,5 bis 2)

Edelson hat Smart Pointer genau analysiert und zeigt die Vor- und Nachteile verschiedener Smart-Pointer-Varianten [Edelson91].
In Abschnitt 5.3 wird der Collector von Detlefs vorgestellt, der mit Smart-Pointern arbeitet.

4.2.3 Manuell erzeugter Code

Alle für einen Collector benötigten Informationen manuell bereitzustellen, ist unzumutbar, weil dies zu aufwendig und fehleranfällig ist. Manche Teilinformationen lassen sich jedoch mit relativ geringem Aufwand manuell codieren.

Dazu gehört auch die Information über die in einem Objekt enthaltenen Pointer. Dazu muß der Programmierer für jede Klasse eine sogennannte Callback-Methode implementieren. Diese Methode wird vom Collector für jedes gefundene Objekt aufgerufen und liefert ihm die enthaltenen Pointer. Hier ein Beispiel:

```
class CResponsive {
     public:
     virtual void Callback() = 0;
};

class Sample : public virtual Responsive {
     Sample* Left;
     Sample* Right;
     public:
     virtual void Callback();
     ...
}

void Sample::Callback() {
     // push all pointers in p on gc-stack
     Stack.Push( &Left );
     Stack.Push( &Right );
}
```

Die Callback-Methode wird in Bartletts Collector und im MSC-Collector verwendet.

4.2.4 Compiler-/Sprachmodifikationen

Sämtliche benötigten Informationen könnten auch vom Compiler erzeugt werden, sogar ohne Änderungen des Sprachstandards.
Um die Root festzulegen, müßte der Compiler eine Tabelle aller statischen Pointer und Objekte erzeugen und für jeden C++-Block einen Stackframe generieren, der Informationen über die Lage der Pointer auf dem Stack liefert. Zur Laufzeit müßte der Compiler beim Aufruf einer Funktion bzw. Eintritt in einen C++-Block zusätzlich zu den automatischen Variablen eine Kennung für den Stackframe auf den Stack legen. Der Stackframe muß auch Informationen darüber enthalten, welche Prozessorregister Adressen enthalten. Bisher ist noch kein Collector implementiert worden, der auf diese Art die Root-Pointer findet.[1]
Damit der Collector die Pointer innerhalb von Objekten finden kann, muß der Compiler für jede Klasse einen Typ-Deskriptor erzeugen, der beschreibt, an welchen Offsets innerhalb des Objekts Pointer stehen. Detlefs hat einen C++-Compiler so modifiziert, daß er solche Deskriptoren erzeugt [Detlefs90].
Einen Vorschlag zur Erweiterung der Sprachdefinition von C++ um Garbage Collection machen Ellis und Detlefs [EllDet93] (s. 5.4).

---

[1] Diwan hat jedoch für Modula-3 einen Compiler so modifiziert, daß er entsprechende Informationen erzeugt [Diwan91]

5 Realisierte Garbage Collectoren für C++

5.1 Der konservative Collector von Boehm

Der Garbage Collector von Boehm [BoeWei88] ist ein konservativer Collector (dieser Ansatz stammt von Boehm/Weiser). Er ist in C geschrieben, eignet sich aber für fast jede Programmiersprache. Zuerst werde ich die einfache Version dieses Collectors beschreiben, dann die generationelle und zuletzt die parallele Version.

5.1.1 Die Allocation

Boehms Collector holt sich vom Betriebssystem 4 KB große Blöcke. Ihre Adressen müssen auf 4KB-Grenzen liegen. Sie werden in einer Blockliste gespeichert.
In jedem Block werden nur Objekte einer Größe gespeichert. Der Block wird in einen Verwaltungs- und einen Objektbereich geteilt. Im Verwaltungsbereich steht die Größe der Objekte, die im Objektbereich gespeichert sind, die Anzahl der schon gespeicherten Objekte und ein Bit-Feld, das für jedes Objekt dieses Blockes ein Tag-Bit für die Mark-Phase enthält.
Wenn der Mutator Speicherplatz für ein Objekt anfordert, wird in der Blockliste und im Verwaltungsbereich des entsprechenden Blockes nachgesehen, ob es ein freies Objekt dieser Größe gibt (Die Größe wird immer auf ein Vielfaches von 4 Byte aufgerundet, damit nicht zu viele verschiedene Blöcke angelegt werden müssen). Wenn ja, wird ein Pointer auf dieses Objekt zurückgeliefert und die Objektanzahl des Blockes inkrementiert. Wenn nein, wird ein neuer Block vom Betriebssystem geholt und initialisiert. Dann wird ein Pointer auf das erste Objekt in diesem Block zurückgeliefert.

5.1.2 Die Collection

Der Collector inspiziert die Prozessorregister, den Stack und die statischen Objekte. Alle inspizierten Daten werden als Adressen interpretiert und es wird getestet, ob sie gültige Pointer auf ein Objekt sind, und zwar folgendermaßen:

1. Wenn die Adresse niedriger ist als die niedrigste Adresse des Heap[1] oder höher als die höchste Adresse des Heap[2]ist, ist sie kein gültiger Pointer.
2. Als nächstes wird geprüft, ob die Adresse innerhalb eines Blocks liegt. Dazu werden die zwölf untersten Bits ausmaskiert. Die entstandene Adresse muß gleich der Adresse eines Blocks sein; wenn diese Adresse nicht in der Blockliste vorkommt, ist die zu prüfende Adresse kein gültiger Pointer.
3. Nun wird geprüft, ob die Adresse auf den Anfang eines Objekts zeigt oder inmitten eines Objekts liegt. Dazu wird von der Adresse die Adresse des Objektbereichbeginns subtrahiert. Wenn der entstandene Wert kein Vielfaches der Objektgröße ist, ist er kein gültiger Pointer. Der Collector setzt voraus, daß zu jedem Objekt, das vom Mutator benutzt wird, ein Pointer auf den Anfang des Objekts existiert.
4. Als letztes wird getestet, ob die Adresse auf ein Objekt in der Free-List zeigt. Wenn ja, ist sie kein gültiger Pointer.

Alle Adressen, die diese vier Tests bestehen, sind gültige Pointer. Die Objekte, auf die sie zeigen, werden daher als erreichbar markiert. Diese Objekte werden wie die Root-Objekte inspiziert, d.h. alle in ihnen enthaltenen Daten werden wie oben als Pointer interpretiert, getestet und ggf. die Objekte, auf die Pointer zeigen, markiert.
Die Collection endet, wenn alle gefundenen Objekte inspiziert worden sind. Alle nicht markierten Objekte sind frei und müssen in die Free-List aufgenommen werden.
Der Algorithmus des konservativen Collectors läßt sich folgendermaßen in C++ formulieren:

```
static CStack Stack;
```

[1] Das ist die Adresse des ersten Objekts des Blocks mit der niedrigsten Adresse aller Blöcke
[2] Das ist die Adresse des letzten Objekts des Blocks mit der höchsten Adresse aller Blöcke

```
void Collect() {
     PushRoots();
     Mark();
     Sweep();
}

Obj* StaticVars;                    // Statische Variablen
Obj* GlobalStack;                   // Programm-Stack

void PushRoots() {
     Obj (*Regs)[RegCount];         // Prozessor-Register

     GetRegs( &Regs );
     PushGuessedPointersIn( Regs );
     PushGuessedPointersIn( StaticVars );
     PushGuessedPointersIn( GlobalStack );
}

void Mark() {
     Obj *p, *q;

     while ( ! Stack.IsEmpty() ) {
          p = Stack.Pop();
          if ( ! p->IsMarked() ) {
               p->Mark();
                    PushGuessedPointersIn( p );
          }
     }
}

void Sweep() {
     Obj* o;

     for ( each object o ) {
          if ( ! o->IsMarked() )
               Free( o );
     }
}
```

5.1.3 Vorteile

- Es ist keinerlei explizite Kooperation des Mutators nötig.
- Der Mutator muß nur eine einzige Bedingung erfüllen: Zu jedem benutzten Objekt existiert ein Pointer auf seinen Anfang, der innerhalb des Zugriffsbereichs des Collectors liegt. Diese Bedingung ist üblicherweise erfüllt.
- Der Collector kann mit anderen Speicherverwaltungen koexistieren.
- Explizite Deallocation ist möglich.
- Der Collector kann jederzeit abgebrochen werden. Dies kann besonders in Verbindung mit opportunistischer Garbage Collection in interaktiven Applikationen genutzt werden.

5.1.4 Nachteile

- Die Mark-Phase dauert wegen der zusätzlichen Tests länger als bei herkömmlichen Mark-and-Sweep-Collectoren.
- Die Möglichkeit einer Fragmentierung des Speichers ist hoch. Boehm selbst berichtet, daß in manchen Situationen etwa die Hälfte des Heap nicht genutzt werden konnte, weil die freien Objekte nicht die vom Mutator benötigte Größe hatten.
- In ungünstigen Fällen wird ein großer Teil der unbenutzten Objekte aufgrund von Fehlinterpretationen unsicherer Pointer nicht freigegeben.
- Bei Verwendung hoch optimierender Compiler ist der Collector nicht zuverlässig, d.h. für zuverlässige Funktionsweise muß die Optimierung abgeschaltet werden (s. 4.2.1).

5.1.5 Der Collector als Debugging-Tool

Für Programme oder Programmteile, die eigentlich explizite Allocation/Deallocation benutzen sollen, kann der Collector folgendermaßen eingesetzt werden, um Storage-Leaks zu entdecken:
Zu jedem allozierten Objekt wird der Name der Funktion gespeichert, die es alloziert hat. Objekte, die explizit freigegeben werden, werden mit Hilfe eines zusätzlichen Free-Bits markiert. Zu einem beliebigen Zeitpunkt wird der Collector aufgerufen. Wenn er Objekte findet, die nicht mehr erreichbar sind, aber deren Free-Bit nicht markiert sind, hat er ein Storage-Leak entdeckt. Er gibt dann die Adresse dieses Objekts und den Namen der Funktion, die es alloziert hat, aus.
Boehm hat mehrere größere Programme auf diese Art getestet. Er berichtet, daß die Fehlermeldungen des Collectors in allen außer zwei Fällen auf echte Storage-Leaks zurückzuführen waren. In den beiden verbleibenden Fällen hatte der Mutator nur noch Pointer auf das Innere des Objekts gehalten, aber keine mehr auf seinen Anfang.
Storage-Leaks sind oft die Ursache von Systemzusammenbrüchen, verursacht von Programmen, die über sehr lange Zeit laufen; sie sind normalerweise äußerst schwer zu finden.

5.1.6 Die generationelle Version

Die generationelle Version des Collectors [DemWeiHay+90] benötigt das Dirty-Bit-Feature der virtuellen Speicherverwaltung. Die Objekte werden in zwei Generationen eingeteilt. Die Objekte, die seit der letzten Collection alloziert wurden, gehören zur neuen Generation. Alle anderen Objekte gehören zur alten Generation. Die Implementation dieser Generationen ist sehr einfach: das Mark-Bit wird vor der Collection nicht gelöscht; alle Objekte mit gesetztem Mark-Bit gehören zur alten Generation. Während der Collection werden diejenigen Seiten, deren Dirty-Bit nicht gesetzt ist, ignoriert. Die Seiten, deren Dirty-Bit gesetzt ist, werden nur während der Mark-Phase berücksichtigt.
Diese Modifikation hat zwar eine Erhöhung des gesamten Zeitbedarfs des Collectors zur Folge, da häufiger eine Collection durchgeführt werden muß. Aber dafür werden erstens die Pausen, die der Collector verursacht, viel kürzer, und zweitens werden bedeutend weniger Seiten vom Collector berührt; dadurch wird das Ein- und Auslagern von Seiten vermieden.

5.1.7 Die (fast) parallele Version

Die parallele Version [BoeDemShe91] nutzt ebenfalls das Dirty-Bit-Feature. Zu Beginn der Collection werden alle Dirty-Bits gelöscht (wenn man die parallele mit der generationellen Version kombinieren will, müssen vorher die Adressen aller Seiten mit gesetztem Dirty-Bit gespeichert werden, weil diese Information von der generationellen Version benötigt wird). Während der Mutator weiterarbeitet, markiert der Collector nun alle erreichbaren Objekte. Sobald er damit fertig ist, stoppt er den Mutator. Nun muß er nur noch die seit Beginn der Collection geänderten Seiten (erkennbar am Dirty-Bit) nachbearbeiten. Dazu nimmt er sie als Root und markiert alle von ihnen aus erreichbaren Objekte. Da üblicherweise nur wenige Seiten geändert sind, verkürzen sich die vom Collector verursachten Pausen erheblich.
Die Sweep-Phase läuft wieder parallel zum Mutator ab.
Der Collector benötigt zwar insgesamt mehr Zeit, merkliche Pausen treten dafür aber praktisch nicht mehr auf.

5.1.8 Blacklisting

Boehm hat aufgrund der Probleme seines Collectors mit Fehlinterpretationen unsicherer Pointer eine Technik entwickelt, die Fehlinterpretationen erheblich seltener macht, das sog. Blacklisting [Boehm93]. Dazu wird in der Mark-Phase jeder unsichere Pointer, der zwar in den Heap-Bereich zeigt, sich aber nach den weiteren Checks als ungültig erweist, in eine Black-List eingetragen. Während der Allocation wird sichergestellt, daß kein Objekt an einer Adresse alloziert wird, die in der Black-List eingetragen ist. Dies wird erreicht, indem eine Seite des Heap nicht verwendet wird, wenn ein Pointer in der Black-List in diese Seite zeigt. Da die virtuelle Speicherverwaltung den Speicher

erst zur Verfügung stellt, wenn tatsächlich auf eine Seite zugegriffen wird, wird durch dieses Vorgehen kein Speicher verschenkt. Der zusätzliche Verwaltungsaufwand für das Blacklisting liegt unter 1% der für Allocation und Collection verbrauchten Laufzeit.

5.2 Der Mostly-Copying-Collector von Bartlett

Der Garbage Collector von Bartlett [Bartlett88] gehört zu den Copy-Collectoren. Bartlett ist es jedoch gelungen, auch mit unsicheren Pointern umzugehen. Er nennt seinen Ansatz **Mostly-Copying-Collection**. Er wurde für einen C-to-Scheme-Compiler entwickelt; es existiert auch eine Version für C++ [Bartlett90a], auf die ich mich im Folgenden beziehe. Es gibt auch eine generationelle Version, die ich später beschreiben werde.

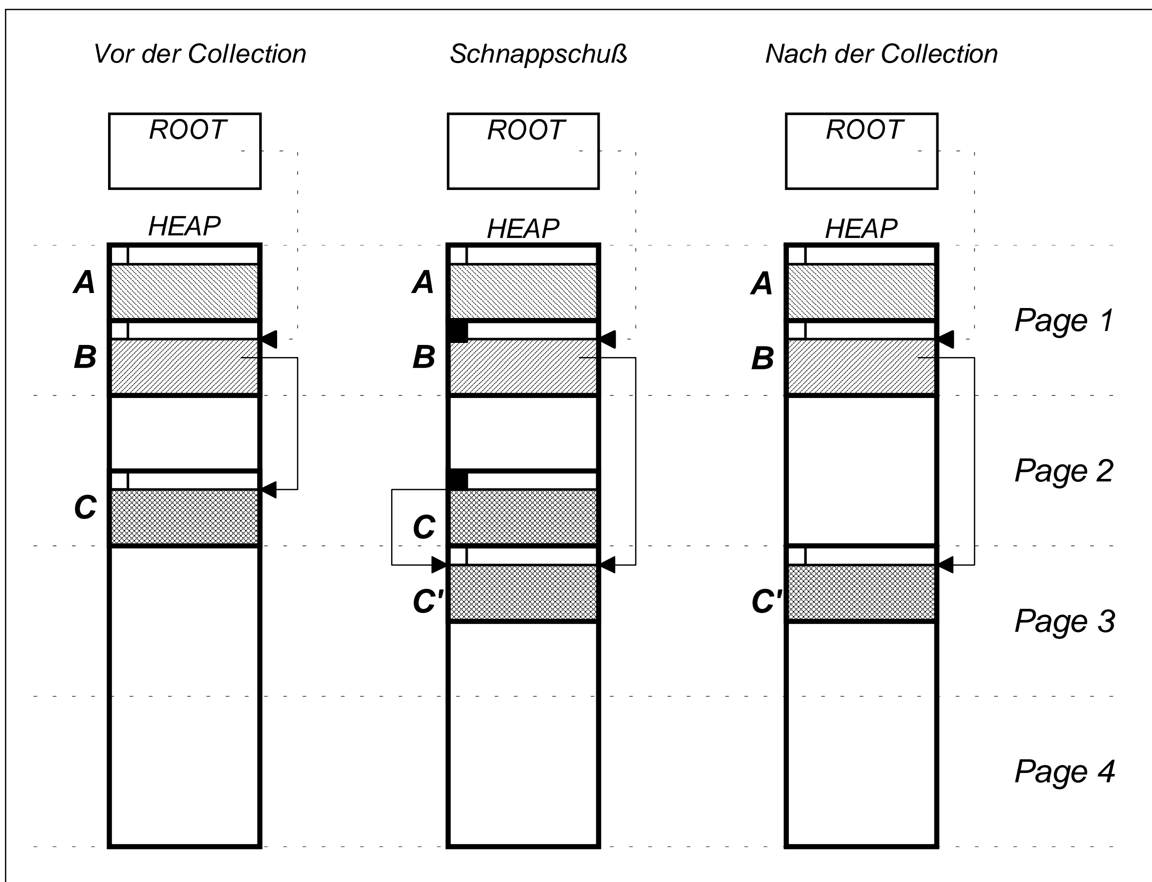
5.2.1 Die Allocation

Zu Beginn fordert der Collector den Speicher für den gesamten Heap vom Betriebssystem an. Er wird, falls erforderlich, nach hinten verlängert, bleibt aber immer zusammenhängend. Der Collector teilt den Heap in Seiten auf (diese Seiten haben nichts mit den Seiten einer virtuellen Speicherverwaltung zu tun). Diese Seiten sind Teil eines sog. Spaces; der Space *i* besteht aus allen Seiten, die vor der *i*-ten Collection mit Objekten belegt worden sind. Zu welchem Space eine Seite gehört, wird in einer für jede Seite vohandenen Variablen *space* gespeichert. Zu Beginn ist *space* für alle Seiten gleich 0.
Im Zähler *current-space* wird die Anzahl der bisher durchgeführten Collectionen gespeichert. Dieser Zähler wird mit 1 initialisiert.
Fordert der Mutator Speicher für ein Objekt an, wird in zwei Schritten verfahren: Zuerst wird eine Seite als belegt gekennzeichnet, indem *space* auf *current-space* gesetzt wird. Dann wird Platz für das Objekt auf dieser Seite reserviert. Der erste Schritt kann entfallen, falls auf der zuletzt belegten Seite noch genügend Platz für das Objekt vorhanden ist. Falls das Objekt nicht mehr auf die angefangene Seite paßt, wird der Rest dieser Seite mit einem Dummy-Objekt belegt und eine neue Seite begonnen. Falls das Objekt länger als eine Seite ist, werden mehrere hintereinanderliegende Seiten als belegt gekennzeichnet und zusätzlich alle außer der ersten Seite als Folgeseiten markiert.
Der Mutator muß bei jeder Anforderung einen Pointer auf eine sogenannte „Callback“-Funktion mitliefern. Diese Funktion wird während der Collection vom Collector aufgerufen und ruft ihrerseits für jeden Pointer des Objekts eine Funktion des Collectors auf.
Vor jedem Objekt wird ein Header angelegt, der die Größe des Objekts und (codiert) die Adresse der Callback-Funktion enthält.

5.2.2 Die Collection

Spätestens kurz bevor 50 % aller Seiten belegt sind, wird eine Collection durchgeführt. Dabei werden die nicht belegten Seiten als to-space verwendet.
Zunächst wird die Variable *next-space* auf *current-space*+1 gesetzt. Verglichen mit dem klassischen Copy-Algorithmes, kann man die Seiten mit *space* = *current-space* als from-Space und die Seiten mit *space* = *next-space* als to-Space betrachten. Als Root werden die Prozessorregister, der Stack und die statischen Variablen verwendet. Die in ihnen enthaltenen Daten sind allerdings unsichere Pointer, d.h. der Collector weiß nicht, ob es sich bei ihnen um Pointer auf Objekte oder um andere Daten handelt. Er interpretiert zunächst alle Daten als Adressen und prüft, ob es sich um Pointer auf Objekte handelt (konservative Methode); nur Adressen auf den Anfang eines Objekts werden als gültige Pointer betrachtet. Es ist jedoch immer noch möglich, daß es sich um ganz andere Daten handelt, die zufällig das gleiche Bitmuster haben wie ein gültiger Pointer. Daher dürfen diese Daten auf keinen Fall geändert werden. Also dürfen auch Objekte, auf die diese Daten zeigen, nicht verschoben werden. Daher behandelt der Collector die Seiten, in denen solche Objekte gespeichert sind gesondert. Anstatt die Objekte in den to-Space zu kopieren, setzt er nur *space* auf *next-space* und bringt so die komplette Seite in den to-Space.

Nun sind genau die Seiten im next-Space, auf deren Objekte Pointer der Root zeigen. Daher kann jetzt die Menge aller Objekte im next-Space als neue Root gewählt werden. Nun wird für jedes solche Root-Objekt die Callback-Funktion aufgerufen. Diese wiederum ruft für jeden Pointer in diesem Objekt eine Funktion des Collectors auf, die folgendes tut: Sie prüft, ob sich das Objekt, auf das der gelieferte Pointer zeigt, bereits im next-space befindet. Das kann sie an einem Tag-Bit im Header jedes Objekts erkennen. Wenn das Objekt sich schon im next-space befindet, wird nur der Pointer auf die neue Adresse umgesetzt.
Im anderen Fall wird es dorthin kopiert. Der dazu benötigte Speicher wird wie oben alloziert mit dem einzigen Unterschied, daß für belegte Seiten *space* auf *next-space* gesetzt wird und nicht auf *current-space*. Die neue Adresse des Objekts wird an seiner alten Adresse abgelegt. Dann wird die Callback-Funktion des kopierten Objekts aufgerufen.
So wird der Graph der erreichbaren Objekte in rekursivem Abstieg durchlaufen, bis ein Blatt des Graphen oder ein schon kopiertes Objekt erreicht ist. Wenn dies für alle Root-Objekte getan worden ist, ist die Collection beendet. Zum Abschluß wird *current-space* inkrementiert.



In der obigen Abb. befinden sich zu Beginn der Collection die Seiten 1 und 2 im *current-space*. Der Collector findet zunächst in der Root den unsicheren Pointer auf B. Da Objekte, auf die unsichere Pointer zeigen, nicht verschoben werden dürfen, ordnet der Collector die ganze Seite 1 dem *next-space* zu. Als nächstes findet der Collector im Objekt B den Pointer auf C. Er kopiert daher C an den Anfang einer unbenutzten Seite (Seite 3) und ordnet diese Seite ebenfalls dem *next-space* zu. Der Pointer in B wird von C auf C' umgesetzt. Zum Schluß werden alle Seiten des *current-space* freigegeben; in diesem Fall ist es nur die Seite 2. Man beachte, das das Objekt A nicht freigegeben werden konnte, weil es in derselben Seite wie das nicht verschiebbare Objekt B liegt.
Der Mostly-Copying Algorithmus in C++-Pseudocode:

```
static CStack Stack;

void Collect() {
    PromoteRootPages();
```

```
    PushObjectsOnPromotedPages();
    Copy();
}
```

```
void PromoteRootPages() {
     Obj* p;
     Page* pPage;

     for ( each root-pointer p ) {
          page = FindPageOfObject( p );
          pPage->Promote();
     }
}

void Copy() {
     Obj** p;

     while ( !Stack.IsEmpty() ) {
          p = Stack.Pop();
          if ( ! (*p)->Copied() || (*p)->Marked() ) {
               if ( IsOnPromotedPage( p ) )
                    (*p)->Mark();
               else {
                    CopyObj( *p );
                    *p = NewAdr( *p );
               }
               (*p)->Callback();
          }
          else
               *p = NewAdr( *p );
     }
}
```

5.2.3 Vorteile

- Der Collector kann auch mit unsicheren Pointern fertigwerden.
- Fragmentierung tritt kaum auf, da größte Teil des Heap bei der Collection verdichtet wird.
- Der Zeitbedarf ist proportional zur Menge des benutzten Speicherplatzes, unabhängig von der Größe des Heap.

5.2.4 Nachteile

- Der Collector kann nur für Programme/Programmteile eingesetzt werden, deren C++-Quelltext verfügbar ist.
- Bei großer Root wird in ungünstigen Situationen nur ein kleiner Teil des unbenutzten Speichers freigegeben.
- Eine Parallelisierung ist nur mit sehr hohem Aufwand möglich.
- Explizite Deallozierung zur Verbesserung der Effizienz des Collectors kann nicht eingesetzt werden, da Objekte grundsätzlich am Ende des bisher belegten Speicherbereichs alloziert werden.
- Der Collector kann nicht unterbrochen oder abgebrochen werden.
- Arrays können nicht gesammelt werden, da sie vom globalen `new` alloziert werden.

5.2.5 Die generationelle Version

Bartlett hat auch eine generationelle Version seines Collectors entwickelt [Bartlett89]. Dabei werden die Objekte in zwei Generationen eingeteilt: Der neuen Generation gehören diejenigen Objekte an, die nach der letzten Collection alloziert wurden; das sind alle Objekte auf Seiten mit *space = current-space*. *current-space* ist zu Anfang 1 und wird nach jeder Collection um 2 erhöht. Dadurch ist *next-space* bei jeder Collection gerade (denn dann ist *next-space = current-space+1*). Die alte Generation besteht aus denjenigen Objekten, die auf Seiten mit geradem *space* liegen. Objekte der alten Generation werden nicht verschoben. Sie werden jedoch immer inspiziert, um die in ihnen enthaltenen Pointer auf Objekte der neuen Generation aktuell zu halten.

5.3 Der Reference-Counting-Collector von Detlefs

Der Collector von Detlefs [Detlefs92] beruht auf der Verwendung von Smart-Pointern. Er ist jedoch ein hybrider Collector, da die Pointer auf dem Stack konservativ gefunden werden. Er ist auch kein reiner Reference-Counting-Collector, denn ein Objekt wird nicht sofort freigegeben, wenn sein Reference-Counter auf 0 geht, sondern es gibt eine besondere Collection-Phase, während der die Objekte freigegeben werden. Detlefs' Smart-Pointer-Implementation gibt diesen Algorithmus jedoch nicht vor; es könnte z.B. auch ein Mark-and-Sweep-Algorithmus verwendet werden.

5.3.1 Die Allocation

In einer Anwendung, die den Collector verwenden will, müssen alle Pointer auf Objekte Smart-Pointer sein. Detlefs hat seine Smart-Pointer als Template implementiert; dadurch muß der Anwender die Smart-Pointer-Klassen nicht mehr deklarieren. Die Deklaration sieht folgendermaßen aus:

```
class PrtAny {
     public:
     PtrAny();
     int operator==( const PtrAny& pa );
};

const PtrAny PtrNil;  //NULL pointer

template<class T> class Ptr: public PtrAny {
     public:
     Ptr();
     void New(); // Allocates new object and makes the Ptr object
                 // point to it

     Ptr( const Ptr<T>& pt );
     Ptr<T>& operator=( const Ptr<T>& pt );

     T& operator*();    // pointer operations
     T* operator->();

     private:
     ...
}
```

In der Anwendung muß überall statt T* nun Ptr<T> verwendet werden.
Objekte werden auf dem Heap alloziert, indem die `New()`-Methode des dazugehörigen Smart-Pointers aufgerufen wird. Statt

```
Sample* p;
p = new Sample;
```

schreibt man nun

```
Ptr<Sample> p;
p.New();
```

Der Smart-Pointer Ptr<T> alloziert auf dem Heap nicht ein Objekt vom Typ T, sondern vom Type Wrapper<T>. Wrapper<T> enthält außer einem Objekt vom Typ T diejenigen Informationen, die der Collector benötigt. Im Fall des Reference-Counting-Collectors ist das u.a. der Reference Counter.

```
class WrapperBase{
     ...
};

template <class T> class Wrapper: public WrapperBase {
     T elem;  // the new object
     public:
     T* GetElem()  // used by Ptr<T> to get the object

     // collector-specific stuff
     int Counter; // the ref counter
     ...
};
```

Diese Technik hat den Vorteil, daß die Klassen des Anwendungsprogramms keine zusätzlichen Methoden o.ä. benötigen.
Mit Hilfe dieser Wrapper ist es auch möglich, von jeder Klasse eine sog. RC-Map zu erzeugen. Das ist eine Tabelle, in der die Lage aller Pointer in einem Objekt verzeichnet ist. Diese Tabelle kann zur Implementation eines Mark-and-Sweep-Collectors verwendet werden.

5.3.2 Die Collection

Der Collector von Detlefs ist im Wesentlichen ein Reference-Counting-Collector. Im Unterschied zum Standard-Algorithmus wird ein Objekt jedoch nicht freigegeben, wenn sein Zähler 0 wird, sondern auf eine sog. Zero Count List (ZCL) gesetzt. Dieses Vorgehen hat zwei Gründe:

- Aufgrund der Eigenarten temporärer Objekte in C++ kann es passieren, daß mitten in einer Pointer-Zuweisung der Zähler des referenzierten Objekts auf Null gesetzt wird (s. [Ginter91]).
- Wenn für Pointer auf dem Stack auf das Reference-Counting verzichtet wird, sinkt nach Detlefs der Laufzeit-Overhead des Collectors signifikant.

Die eigentliche Collection ist Mark-and-Sweep ähnlich: Zunächst werden die Register und der Stack mit der konservativen Methode nach Pointern durchsucht. Alle Objekte in der ZCL, auf die ein Pointer in einem Register oder auf dem Stack zeigt, werden markiert. Dieses Markieren muß nicht rekursiv durchgeführt werden, da diejenigen Objekte, die von einem Objekt auf der ZCL benutzt werden, einen Zähler > 0 haben.
Anschließend werden alle Objekte auf der ZCL, die nicht markiert sind, freigegeben.

5.3.3 Vorteile

- Der Anwender muß keine Information über den Aufbau seiner Klassen/Objekte zur Verfügung stellen.
- Die gleichzeitige Verwendung von Klassen, deren Objekt gesammelt werden, und solcher, die nicht gesammelt werden, ist möglich.
- Der Destruktor von freigegebenen Objekten wird aufgerufen.
- Die Pausen durch Collection sind sehr kurz (unter 0,1 Sek.)

5.3.4 Nachteile

- Smart-Pointer haben nicht alle Eigenschaften von C++-Pointern (s. 4.2.2)
- Zyklische Strukturen können nicht freigegeben werden
- Die Performance ist sehr schlecht. Detlefs hat in seinen eigenen Experimenten eine Erhöhung der Laufzeit um den Faktor 1,6 bis 2 festgestellt.

5.4 Das Proposal von Ellis/Detlefs

Das Proposal von Ellis/Detlefs [EllDet93] fällt insofern aus dem Rahmen, als Ellis und Detlefs keinen entsprechenden Collector gebaut haben. Dennoch lohnt es sich, sich mit ihrem Proposal zu beschäftigen, weil es als erster versucht, das Problem Garbage Collection in C++ komplett zu behandeln, während die anderen Collectoren nur Teillösungen darstellen. Das Proposal besteht aus mehreren Teilen:

- Einer Erweiterung der Sprache
- Einem Subset der Sprache, das so ausgelegt ist, daß keine Fehler mehr aufgrund von Pointer-Operationen vorkommen können (wie z.B. dangling references)
- Für C++ geeigneten Garbage-Collection-Algorithmen
- Anforderungen zur Codeerzeugung an den Compiler

Auf den zweiten Punkt werde ich nicht eingehen, weil er mit den Problemen von Garbage Collection nur indirekt zu tun hat.

5.4.1 Die Spracherweiterung

Es werden zwei Schlüsselworte eingeführt, mit denen Programmierer kennzeichnen können, ob die Objekte einer Klasse gesammelt werden sollen oder nicht. Objekte einer Klasse, die mit

```
gc class A {...}
```

deklariert wird, werden vom Collector gesammelt. Klassen, die mit

```
nogc class A {...}
```

deklariert werden, werden nicht gesammelt. Um die Kompatibilität zu gewährleisten, werden Klassen ohne eins der neuen Schlüsselwörter ebenfalls nicht gesammelt.
Klassen, die mit **gc** deklariert werden, dürfen new und delete nicht überladen. Für sie werden die entsprechenden Routinen des Collectors aufgerufen.

5.4.2 Die Collector-Algorithmen

Ein wichtiges Anliegen von Ellis/Detlefs ist, Garbage Collection auch mit schon bestehenden Bibliotheken zu ermöglichen, ohne diese ändern zu müssen. Daher muß der Collector mindestens teilweise konservativ arbeiten. Obwohl sich die Autoren nicht festlegen, scheinen sie deutlich einen Collector zu bevorzugen, der auf die gleiche Weise arbeitet wie Boehms Collector. Es soll zwei Heaps geben, einen für die GC-Objekte, einen für die Non-GC-Objekte. Der Non-GC-Heap wird als Teil der Root behandelt, sodaß auch GC-Objekte, die an Non-GC-Funktionen übergeben worden sind, korrekt als benutzt erkannt werden.

5.4.3 Der Destruktoraufruf

Grundsätzlich wird für alle freigegebenen Objekte der Destruktor aufgerufen, falls er deklariert worden ist. Dies kann jedoch mit Hilfe eine vom Collector zur Verfügung gestellten Routine abgeschaltet werden. Das Problem des Destruktoraufrufs bei zyklisch verketteten Objekten wird "gelöst", indem solche Objekte nicht freigegeben werden. Dies ist ein gravierender Nachteil, den der Collector mit Reference-Counting-Collectoren gemeinsam hat.
Da Ellis und Detlefs die konservative Methode als Collector-Algorithmus bevorzugen, ist nicht garantiert, daß alle unbenutzten Objekte freigegeben und damit auch destruiert werden.

5.4.4 Weak Pointer

Für viele Anwendungen ist es nötig, Pointer zu haben, die für den Collector nicht als benutzende Referenzen zählen, wie z.B. für Caches. Für diesen Fall haben Ellis und Detlefs eine spezielle Konstruktion vorgesehen, und zwar das Template WeakPointer<T>. Ein solcher Pointer liefert den Pointer auf das Objekt zurück, falls es noch nicht freigegeben worden ist, sonst NULL. Der Collector prüft bei jedem unbenutzten Objekt, ob Weak Pointer darauf zeigen, und setzt diese auf NULL, bevor er das Objekt freigibt.

5.4.5 Anforderungen an die Codeerzeugung

Ellis/Detlefs gehen davon aus, daß die Register und der Stack auf jeden Fall mit der konservativen Methode durchsucht werden. Daraus ergeben sich Probleme, wenn der Compiler so optimiert, daß keine Adresse auf das noch benutzte Objekt mehr existiert. Ein Beispiel dafür findet sich in Kap. \ref. Die Autoren zeigen, daß der Compiler solche Situationen leicht vermeiden kann, und daß der dadurch entstehende Optimierungsverlust marginal ist.

5.5 Der Mark-Sweep-Copy-Collector

Der Mark-Sweep-Copy-Collector ist im Rahmen dieser Studienarbeit prototypisch implementiert worden. Es liegen jedoch noch keine Erfahrungen über die Verwendung in einer größeren Anwendung vor.

5.5.1 Die Schnittstelle

Die Schnittstelle des Mutators zum Mark-Sweep-Copy-Collector besteht aus vier Teilen:

- Jede Klasse, deren Objekte gesammelt werden sollen, erbt von der Klasse `BaseObj`. `Base-Obj` muß die Wurzel des Vererbungsbaumes sein oder die jetzige Wurzel muß entsprechend ergänzt werden.
- Die Konstruktoren und Destruktoren dieser Klasse werden wie in \ref{Adressenzuordnung} beschrieben ergänzt.
- Für jede dieser Klassen wird eine `CallbackPointers()`-Elementfunktion implementiert.
- Das globale `new` und `delete` wird überladen (`new` und `delete` dürfen für die einzelnen sammelbaren Klassen nicht mehr vom Mutator überladen werden).

An welchen Stellen diese Maßnahmen benötigt werden, wird in den folgenden Abschnitten beschrieben.

5.5.2 Die Allocation

Heap-Objekte

Bei der Allozierung eines Objekts reserviert `new` zunächst den nötigen Speicherplatz, indem es `malloc()`aufruft. Dann wird der zurückgelieferte Speicherblock in die Blockliste eingetragen. Nun liefert `new` die Adresse des allozierten Blocks zurück.
Als nächstes werden die Konstruktoren für das neue Objekt aufgerufen, als erstes der `BaseObj`-Konstruktor. Er trägt das Objekt in die Objekt-Liste ein. Alle anderen Konstruktoren rufen `Regi-sterPointer()` auf. Diese Funktion erhält als Parameter die Adresse, die als Adresse des neuen Objekts zu interpretieren ist (`(void *)this`) und die dazugehörige `BaseObj`-Referenz (`(BaseObj *)this`). Diese Angaben werden in die Adressen-Liste eingetragen.

Non-Heap-Objekte

Für Non-Heap-Objekte werden natürlich ebenfalls die Konstruktoren aufgerufen. Der Collector erkennt, daß sie nicht mit `new` alloziert worden sind und fügt ihre Pointer der Root hinzu.

Pointer

Globale Pointer müssen mit der Funktion `RegisterRootPointer()` beim Collector angemeldet werden. Diese Pointer werden der Root hinzugefügt. Lokale Pointer werden mit der konservativen Methode gefunden.

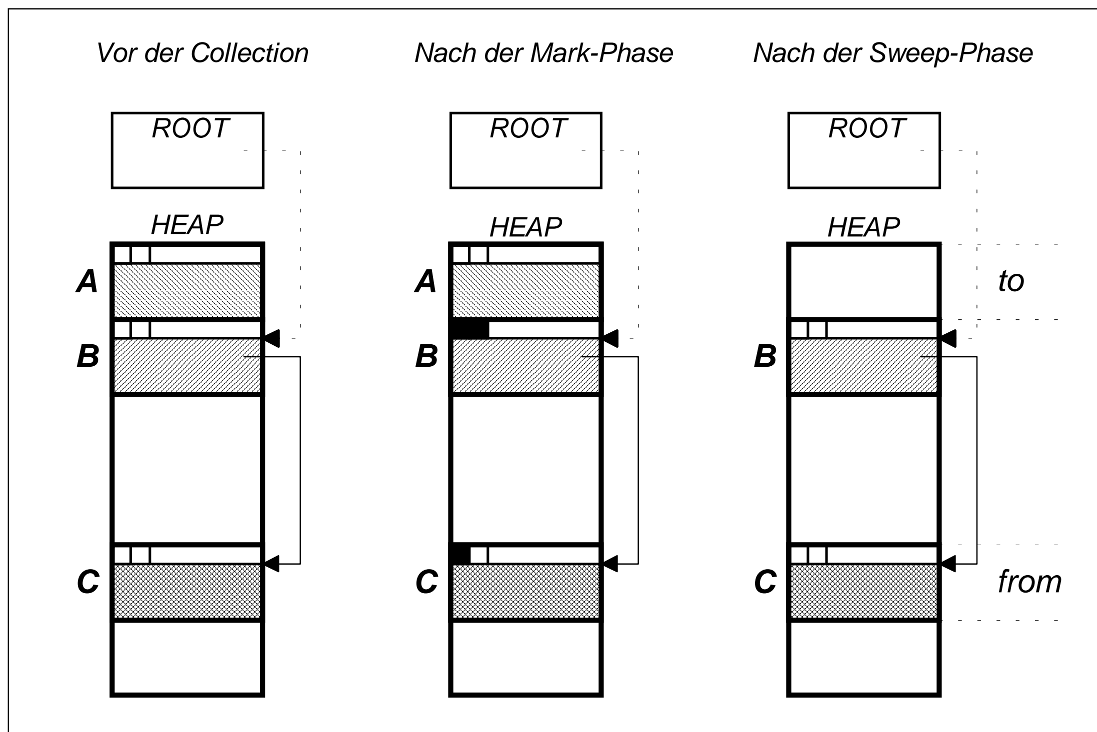
5.5.3 Die Collection

Bisher sind nur die Mark- und die Sweep-Phase implementiert worden. Welche zusätzlichen Schritte für die Einführung einer Copy-Phase nötig wären, wird jeweils als Anmerkung beschrieben.

Die Mark-Phase

Für die Mark-Phase wird ein Collector-Stack eingerichtet. Als erstes werden die Root-Pointer auf den Stack gepusht. Für die globalen Pointer und die Root-Objekte ist das trivial. Für die Pointer im Stack wird nach der konservativen Methode vorgegangen: Jedes in Frage kommende Bitmuster wird in der Adressen-Liste gesucht. Ist das Bitmuster als gültige Objektadresse interpretierbar, wird der in der Adressen-Liste mitgespeicherte BaseObj-Pointer auf den Collector-Stack gepusht.
Die eigentliche Mark-Phase läuft folgendermaßen ab:
Der oberste Objekt-Pointer wird vom Collector-Stack gepopt. Ist das dazugehörige Objekt schon markiert, ist nichts mehr zu tun. Im anderen Fall wird das Objekt markiert und dann seine Callback()-Funktion aufgerufen. Die zurückgelieferten Pointer werden getestet und, falls es Objekt-Pointer sind, auf den Collector-Stack gepusht.
Dies wird solange wiederholt, bis der Collector-Stack leer ist. Dann ist die Mark-Phase beendet.
Anmerkung: Für die Durchführung der Copy-Phase müßte auf dem Collector-Stack zusätzlich die Herkunft jedes Pointers abgelegt werden; denn Pointer, die auf dem Stack mit der konservativen

Methode gefunden werden, sind unsichere Pointer. Die Objekte, auf die sie zeigen, dürfen daher während der Copy-Phase nicht verschoben werden. Sie müssen also auch entsprechend markiert werden.



Im Beispiel (obige Abb.) untersucht der Collector als erstes die Root und findet den unsicheren Pointer auf B. B wird daher nicht nur als benutzt markiert, sondern auch als nicht verschiebbar gekennzeichnet (zweites schwarzes Quadrat in der linken oberen Ecke des Objekts). Beim Untersuchen von B findet der Collector den Pointer auf C und markiert C als benutzt.

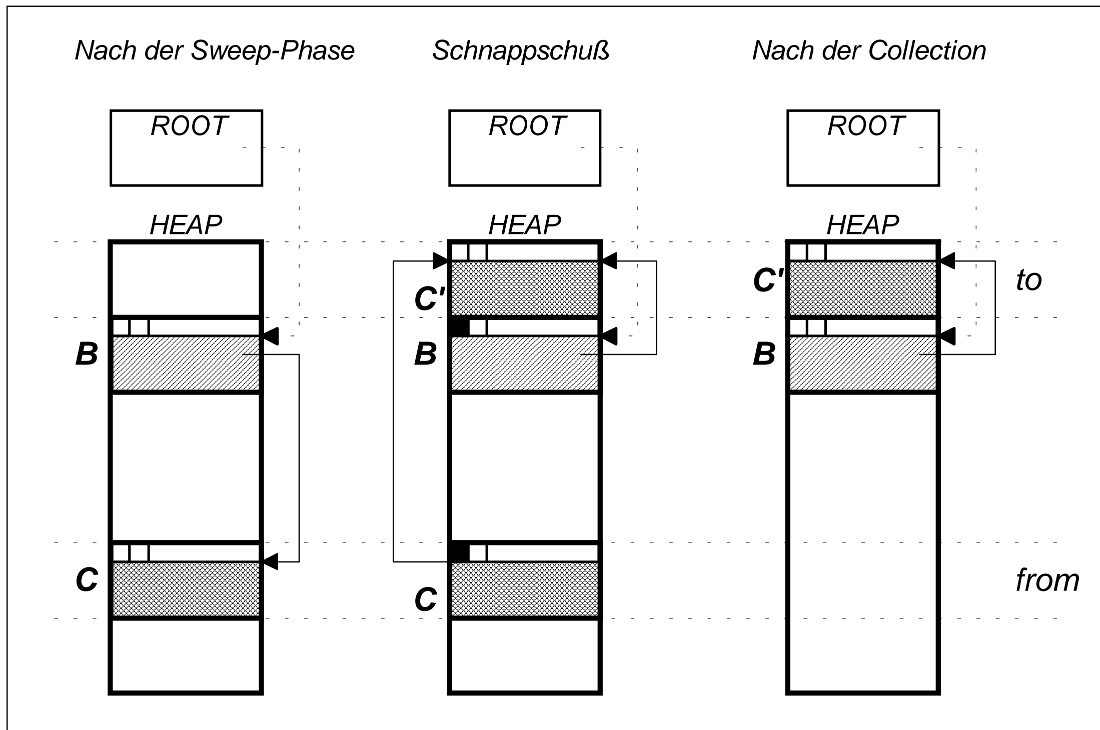
Die Sweep-Phase

In der Sweep-Phase wird jedes Objekt in der Objekt-Liste behandelt: Ist es markiert, wird die Markierung gelöscht; ist es nicht markiert, wird für dieses Objekt `delete` aufgerufen.
Anmerkung: Für die Durchführung der Copy-Phase muß während der Sweep-Phase mitprotokolliert werden, auf welchen Adressen nicht-verschiebbare Objekte liegen. Anhand dieser Informationen müßte er dann den optimalen from- und to-space bestimmen.
Im Beispiel wird während der Sweep-Phase das Objekt A freigegeben. Außerdem merkt sich der Collector, welche Abschnitte frei sind (in diesem Fall u.a. der Platz von A) und welche Objekte verschoben werden dürfen (in diesem Fall C). Mittels dieser Informationen bestimmt der Collector nach der Sweep-Phase den from-space (nur Objekt C) und den to-space (der Platz von Objekt A) für die Copy-Phase.

Die Copy-Phase

Die Copy-Phase dient nicht zum Sammeln unbenutzter Objekte, denn nach Durchführung der Sweep-Phase sind ja keine solchen Objekte mehr vorhanden. Sie muß nicht bei jeder Collection durchgeführt werden, da sie nur dazu dient, den fragmentierten Speicher wieder zu kompaktieren. Dazu wird zunächst mit Hilfe der in der Sweep-Phase gesammelten Informationen ausgerechnet, welche Objekte wohin verschoben werden sollen, um möglichst wenig Lücken freien Speichers zwischen den benutzten Objekten zu behalten.
Die eigentliche Copy-Phase ist der Mark-Phase sehr ähnlich. Nur wird diesmal nicht mit den Pointern auf Objekte, sondern mit Pointern auf Pointer gearbeitet:

Zuerst werden die Adressen aller Root-Pointer auf den Collector-Stack gepusht. Dann wird folgender Schritt so oft wiederholt, bis der Collector-Stack leer ist:Der oberste Pointer wird vom Collector-Stack gepopt. Ist das Objekt, auf das dieser Pointer indirekt zeigt, schon kopiert, wird der Objektpointer auf die neue Adresse gesetzt. Zeigt der gepopte Pointer auf einen unsicheren Pointer und dieser auf ein noch umarkiertes Objekt, wird das Objekt markiert und die Adressen der in ihm enthaltenen Pointer auf den Collector-Stack gepusht. Zeigt er dagegen auf einen sicheren Pointer und ist das Objekt noch nicht markiert, so wird es kopiert und an seiner alten Adresse seine neue abgelegt; außerdem werden wie im ersten Fall die Adressen der in ihm enthaltenen Pointer auf den Collector-Stack gepusht.
Nun ist die Copy-Phase beendet und die Fragmentierung so gut wie möglich beseitigt.



In der obigen Abb. ist die Copy-Phase des Beispiels zu sehen. Während dieser Phase wird das Objekt C kopiert (nach C') und der Pointer in B von C auf C' umgesetzt (mittleres Bild). Am Ende der Copy-Phase wird der from-space freigegeben.
Bei der Betrachtung des folgenden C++-Pseudocodes werden die Ähnlichkeiten des Mark-Sweep-Copy-Collectors zum konservativen und zum Mostly-Copying Collector deutlich:

```
static CStack UnsafeStack, SafeStack;

void Collect() {
     DeleteMarkBits();
     PushRootUnsafe();
     Mark();
     SweepAndChooseFromSpace();
     if ( CopyingNeeded ) {
          PushRootUnsafe();
          Copy();
     }
}

void Mark() {
     Obj** p;

     while ( !UnsafeStack.IsEmpty() || !SafeStack.IsEmpty() ) {
          if ( ! SafeStack.IsEmpty() ) {
               *p = SafeStack.Pop();
               if ( !(*p)->IsMarked() ) {
```

```
                    (*p)->Mark();
                    if ( IsResponsive( *p ) )
                         (*p)->Callback();
                    else
                         PushGuessedPointersUnsafe();
          }
          else {
               *p = UnsafeStack.Pop();
               if ( !(*p)->IsMarked() ) {
                    (*p)->MarkAsFixed();
                    if ( IsResponsive( *p ) )
                         (*p)->Callback();
                    else
                         PushGuessedPointersUnsafe();
               }
               else {
                    if ( ! (*p)->IsFixed() )
                    (*p)->MarkAsFixed();
               }
          }
     }
}

void Copy() {
     Obj** p;

     while ( !UnsafeStack.IsEmpty() || !SafeStack.IsEmpty() ) {
          if ( ! SafeStack.IsEmpty() ) {
               *p = SafeStack.Pop();
               if ( !(*p)->IsMarked() ) {
                    if ( IsInFromSpace(*p) ) {
                         CopyObj( *p );
                         (*p)->MarkAsCopied();
                         *p = NewAddress( *p );
                    }
                    else {
                         (*p)->Mark();
                    }
                    if ( IsResponsive( *p ) )
                         (*p)->Callback();
                    else
                         PushGuessedPointersUnsafe();
               }
               else {
                    if ( (*p)->IsCopied() )
                         *p = NewAdr( *p );
               }
          }
          else {
               *p = UnsafeStack.Pop();
               if ( !(*p)->IsMarked() ) {
                    (*p)->Mark();
                    if ( IsResponsive( *p ) )
                         (*p)->Callback();
                    else
                         PushGuessedPointersUnsafe();
               }
          }
     }
}
```

5.5.4 Vorteile

- Explizite Kooperation des Mutators ist nicht für alle Klassen erforderlich; daher kann der Collector zusammen mit Bibliotheken eingesetzt werden, deren Quelltext nicht zur Verfügung steht.
- Explizite Deallocation ist möglich.

- Der Collector kann während der Mark- und Sweep-Phase jederzeit abgebrochen werden.
- Der Collector kann zur Entdeckung von Storage-Leaks eingesetzt werden.
- Die Fragmentierung des Speichers ist so gering wie möglich.
- Die Mark-Phase wird durch die Einführung der Callback-Methode schneller als beim Conservative-Collector.

5.5.5 Nachteile

- Eine komplette Collection dauert etwa doppelt so lange wie beim Conservative-Collector; sie ist aber nur selten erforderlich.
- Der Implementierungsaufwand ist etwa doppelt so hoch wie bei üblichen Collectoren.

5.5.6 Ausblick

Der Mark-Sweep-Copy-Collector läßt auf mehrere Weisen weiterentwickeln:

- **Parallelisierung**: Ohne großen Aufwand könnte man die Sweep-Phase parallel zum Mutator ablaufen lassen. Eine weitergehende Parallelisierung wäre in Verbindung mit virtuellem Speicher wie in 3.2 beschrieben möglich
- **Generationelle Collection**: Generationelle Collection könnte implementiert werden, indem in der Copy-Phase „alte“ Objekte in einen besonderen Speicherbereich kopiert werden, in dem nur noch bei einer vollen Collection gesammelt wird.

5.6 Vergleichende Untersuchungen

Untersuchungen, die die Effizienz verschiedener Collectoren vergleichen, sind rar. Eine dieser Untersuchungen hat Hartel durchgeführt [Hartel90]. Er maß den Zeitverbrauch eines Mark-and-Sweep-, eines Copy- und eines Reference-Counting-Collectors und stellte fest, daß der Mark-and-Sweep-Collector im Durchschnitt etwas schneller als der Copy-Collector war. Der Reference-Counting-Collector war dann schneller, wenn der zur Verfügung stehende Heap nur knapp für den Speicherbedarf des Mutators reichte. War der Heap dagegen doppelt so groß wie der Speicherbedarf, nahmen die beiden anderen Collectoren die vorderen Plätze ein.

Zorn verglich den Zeit- und Speicherverbrauch eines generationellen Mark-and-Sweep-Collectors mit dem eines generationellen Copy-Collectors [Zorn90]. Das Ergebnis war, daß der Mark-and-Sweep-Collector im Durchschnittt ca. 20\% weniger Speicher verbrauchte, aber etwa 5\% mehr Rechenzeit.

Dabei ist zu berücksichtigen, daß sowohl Hartel als auch Zorn nur Objekte einer Größe sammeln ließen. In diesem Spezialfall tritt die bei Mark-and-Sweep-Collectoren gefürchtete Fragmentierung des Speichers nicht auf.

Auch Bartlett hat eine vergleichende Untersuchung durchgeführt, deren Ergebnisse in die gleiche Richtung weisen wie die Zorns und Hartels [Bartlett90b]. Er verglich Boehms konservativen Collector mit seinem Mostly-Copying-Collector. In seinem ersten Experiment kamen zu 99\% nur Objekte in drei verschiedenen Größen vor. In diesem Fall war der Collector von Boehm schneller als Bartletts Collector.

Wenn jedoch Objekte mit vielen unterschiedlichen Größen vorkamen (zweites Experiment), wurde der konservative Collector aufgrund der auftretenden Fragmentierung erheblich langsamer. Besonders negativ fiel auf, daß er infolge der starken Fragmentierung mehr als doppelt so viele Seiten der virtuellen Speicherverwaltung mit Objekten belegte wie der Mostly-Copying-Collector.

Beide Collectoren waren, wenn ein etwa dreimal so großer Heap zur Verfügung stand wie mindestens für den Mutator nötig gewesen wäre, etwa genauso schnell wie explizite Allozierung/Deallozierung. Reference Counting kostete dagegen etwa viermal so viel Zeit wie explizite Allozierung/Deallozierung.

ANHANG

A Glossar

Allocation	Die Bearbeitung einer Speicheranforderung des Mutators durch den Collector
Collection	Das Finden von nicht mehr erreichbaren Objekten und die erneute Bereitstellung ihres Speichers.
Collector	Der Prozeß oder der Programmteil, der für die dynamische Speicherverwaltung zuständig ist (Allocation und Collection)
Copy-Collector	Ein Collector, der die erreichbaren Objekte in einen eigens reservierten Speicherbereich (to-Space) kopiert
Dirty-Bit	Ein Bit, das für jede Seite des Hauptspeichers existiert und angibt, ob ein Schreibzugriff auf diese Seite erfolgt ist.
Erreichbares Objekt	Objekt, das über Pointer mit einem Objekt in der Root verkettet ist.
Free-List	Liste von Objekten, die nicht benutzt werden. Bei einer Allocation wird diese Liste nach einem Objekt passender Größe durchsucht und dieses zurückgegeben.
From-Space	(Copy-Collector) Der Speicherbereich, von dem die erreichbaren Objekte kopiert werden
Gefundenes Objekt	(Collection) Objekt, von dem der Collector bereits erkannt hat, daß es erreichbar ist
Generationelle GC	Ansatz der GC, in der meistens nur ein Teil der Objekte inspiziert wird, um Zeit zu sparen. Die Objekte werden dazu in Generationen eingeteilt. Nur die jüngste Generation wird gesammelt. Manchmal muß auch eine volle Collection durchgeführt werden.
Heap	Der Teil des Speichers, der vom Collector verwaltet wird.
Konservative GC	Mark-and-Sweep-Collection, in der als Root die Prozessorregister, der Stack und die statischen Variablen gewählt werden. Alle Daten werden als Pointer interpretiert. Dadurch sind keine Informationen über die innere Struktur der Objekte erforderlich.
Mark-Phase	(Mark-and-Sweep-Collection) Phase, in der alle erreichbaren Objekte markiert werden.
Mark&Sweep-GC	Collection, in der die nicht mehr erreichbaren Objekte in eine Free-List eingefügt werden. Wird in zwei Phasen durchgeführt (Mark-Phase, Sweep-Phase)
Mostly-Copying-GC	Copy-Collection, in der die Objekte, auf die unsichere Pointer zeigen, nicht kopiert werden.
Mutator	Der Prozeß oder der Programmteil, der die dynamische Speicherverwaltung benutzt.
Non-Stop-Collector	Collector, der den Mutator während der Collection nicht anhält.
Opportunistische GC	Ansatz der GC, die Collection zu für den Mutator möglichst günstigen (opportunen) Zeitpunkten durchzuführen.

Reference-Counting Für jedes Objekt wird die Anzahl der Pointer, die auf es zeigen, in einem Zähler gespeichert. Objekte mit einem Zähler >0 sind erreichbar (Ausnahme: unbenutzte zyklische Strukturen); Objekte mit einem Zähler =0 sind frei und dürfen erneut alloziert werden.

Root Die Objekte, von denen der Collector ausgeht, um alle erreichbaren Objekte zu finden.

Stabile Menge (Generationelle GC) Die Menge der Objekte, die nicht in die Collection einbezogen werden.

Storage Leak (Manuelle Speicherverwaltung) Ein Programm, das Speicherbereiche, die es angefordert hat, nicht wieder freigibt, hat ein Storage Leak.

Sweep-Phase (Mark-and-Sweep-Collection) Phase, in der alle nicht markierten Objekte in die Free-List aufgenommen werden.

To-Space (Copy-Collector) Speicherbereich, in den während der Collection alle erreichbaren Objekte kopiert werden.

Unsichere Pointer (Konservativer GC) Daten, deren Bitmuster identisch mit dem Bitmuster einer Adresse eines Objekts sind, von denen der Collector aber nicht weiß, ob es tatsächlich ein Pointer auf ein Objekt ist oder die Bitmusteridentität zufällig ist.

Volle Collection (Generationelle GC) Eine Collection, in der alle Objekte inspiziert werden, nicht nur die der jungen Generation.

B Bibliographie

Die im Textteil referenzierten Artikel sind mit einem * markiert. Um die Suche zu erleichtern, sind die Artikel mit folgenden Kürzeln versehen:

SUR	Überblicksartikel
C++	GC in C++
GEN	Generationelle GC
PAR	Parallele bzw. Concurrent GC
INC	Inkrementelle GC
RT	Real-Time GC
FIN	GC und Object Finalization
PER	Performance-Untersuchungen
CMP	Vergleichende Untersuchung
CON	Konservative GC
RC	Reference Counting GC
DIS	GC in verteilten Systemen

Zahlreiche weitere Literaturhinweise liefern [Cohen81], [Sankaran94] und [Wilson94].

[Appel89]* GEN
Andrew W. Appel.
Simple Generational Garbage Collection and Fast Allocation.
In Software - Practice and Experience, 19(2):171-183, 1989.

[AppEllLi88] * PAR RT
Andrew W. Appel, John R. Ellis und Kai Li.
Real-time Concurrent Collection on Stock Multiprocessors.
In Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 23(7):11-20, 1988.

[AttFla94] FIN
Giuseppe Attardi und Tito Flagella.
Customising Object Allocation.
In ???, :320-343, 1994
net: {attardi, tito}@di.unipi.it

[BarZor93] GEN
David A. Barrett and Benjamin Zorn.
Using Lifetime Predictors to Improve Memory Allocation Performance.
In Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 28(6):187-196, 1993

[Bartlett88] * C++
Joel F. Bartlett.
Compacting Garbage Collection with Ambiguous Roots.
In LISP Pointers 1(6):3-12, 1988.

[Bartlett89] * C++
Joel F. Bartlett.
Mostly Copying Garbage Collection Picks Up Generations and C++.
Technischer Report TN-12, DEC Western Research Laboratory, 1989.

[Bartlett90a] * C++
Joel F. Bartlett.
A Generational, Compacting Garbage Collector for C++.
In Workshop: Garbage Collection in Object-Oriented Systems, 1990.
Available at ftp.diku.dk:/pub/GC90.

[Bartlett90b]*	C++	Joel F. Bartlett. C++ Quelltext des Mostly Copying Garbage Collectors. Technischer Report, DEC Western Research Laboratory, 1990.
[Boehm91]*	CON C++	Hans-J. Boehm. Simple GC-Safe Compilation. In Workshop: Garbage Collection in Object-Oriented Systems, 1991. Available at ftp.diku.dk:/pub/GC91. net: boehm@xerox.com
[Boehm93]*	CON C++	Hans-J. Boehm. Space Efficient Conservative Garbage Collection. In Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 28(6):197-206, 1993
[BoeDemShe91]*	CON PAR	Hans-Juergen Boehm, Alan Demers und Scott Shenker. Mostly Parallel Garbage Collection. In Proceedings of the SIGPLAN '91 Conference on Programming Languages and Implementation, SIGPLAN Notices, 26(6):157-164, 1991.
[BoeWei88]*	CON	Hans-Juergen Boehm und Mark Weiser. Garbage Collection in an Uncooperative Environment. In Software - Practice and Experience, 18(9):807-820, 1988.
[Caplinger88]	C++	Michael Caplinger. A Memory Allocator with Garbage Collection for C. In USENIX Winter Conference Proceedings, Seiten 325-330, 1988.
[CarMatPob+91]	ALG	Svante Carlsson, Christer Mattson, Patricio V. Poblete und Mats Bengtson. A New Compacting Garbage-Collection Algorithm with a Good Average-Case Performance. In STACS'91. 8th Annual Symposium on Theoretical Aspects of Computer Science Proceedings, Band 480, Seiten 296-308, 1991.
[Chambers91]	PER	Cost of Garbage Collection in the SELF System. In Workshop: Garbage Collection in Object-Oriented Systems, 1991. Available at ftp.diku.dk:/pub/GC91. net: chambers@cs.washington.edu
[Cohen81]*	SUR	Jacques Cohen. Garbage Collection of Linked Data Structures. In ACM Computing Surveys, 13(3):341-367, 1981.
[Creak91]*		G. Alan Creak. Garbage - Further Investigations. In SIGPLAN Notices, 26(10):9-10, 1991.
[DemWeiHay+ 90]*	CON GEN	Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow und Scott Shenker. Combining Generational and Conservative Garbage Collection: Framework and Implementations. In Conference Record of the Seventeenth ACM Symposium on Principles of Programming Languages, Seiten 261-269, 1990.

[Detlefs90]* C++ PAR Daniel L. Detlefs.
Concurrent Garbage Collection for C++. Technischer Report, Carnegie Mellon University, 1990.

[Detlefs92]* C++ Daniel L. Detlefs.
Garbage Collection and Run-time Typing as a C++-Library.
In USENIX C++ Technical Conference Proceedings, :37-56, 1992
net: detlefs@src.dec.com

[Diwan91]* Amer Diwan.
Stack Tracing In A Statically Typed Language.
In Workshop: Garbage Collection in Object-Oriented Systems, 1991.
Available at ftp.diku.dk:/pub/GC91.

[DiwTarMos94] PER Amer Diwan, Dvid Tarditi and Eliot Moss.
Memory Subsystem Performance of Programs Using Copying Garbage Collection.
In POPL-1/94???, :1-14, 1994
net: diwan@cs.umass.edu, dtarditi@cs.cmu.edu, moss@cs.umass.edu

[DolGon94] PAR Damien Doligez and Georges Gonthier.
Portable, Unobtrusive Garbage Collection for Multiprocessor Systems.
In POPL-1/94???, :70-83, 1994
net: damien.doligez@inria.fr, georges.gonthier@inria.fr

[DybBruEby93]* FIN R. Kent Dybvig, Carl Bruggeman und David Eby.
Guardians in a Generation-Based Garbage Collector.
In Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 28(6):207-216, 1993
net: {dyb, bruggema, deby}@cs.indiana.edu

[Edelson91]* C++ D. R. Edelson
Smart Pointers: They're Smart, but They're Not Pointers
In Proceedings of the 1991 Usenix C++ Conference, 1-19, 1991
net: edelson@sor.inria.fr

[EdePoh90] C++ D. R. Edelson und I. Pohl.
The Case for Garbage Collection in C++.
In Workshop: Garbage Collection in Object-Oriented Systems, 1990.
Available at ftp.diku.dk:/pub/GC90.

[EllDet93]* C++ John R. Ellis und David L. Detlefs.
Safe, Efficient Garbage Collection for C++.
Unpublished Draft, 1993
net: ellis@parc.xerox.com, detlefs@src.dec.com

[EngVan91] RT Steven L. Engelstad und James E. Vandendorpe.
Automatic Storage Management for Systems with Real Time Constraints.
In Workshop: Garbage Collection in Object-Oriented Systems, 1991.
Available at ftp.diku.dk:/pub/GC91.

[Ferreira91] C++ Paulo Ferreira.
Garbage Collection in C++.
In Workshop: Garbage Collection in Object-Oriented Systems, 1991.
Available at ftp.diku.dk:/pub/GC91.

[FurMatYon91]	CON PAR	Shinichi Furusou, Satoshi Matsuoka und Akinori Yonezawa. Parallel Conservative Garbage Collection with Fast Object Allocation. In Workshop: Garbage Collection in Object-Oriented Systems, 1991. Available at ftp.diku.dk:/pub/GC91.
[Ginter91]*	C++	Cooperative garbage collectors using smart pointers in the C++ programming language. Master's thesis, University of Calgary, 1991
[Hartel90]*	CMP	Pieter H. Hartel. A Comparison of Three Garbage Collection Algorithms. In Structured Programming, 11(3):117-127, 1990.
[Hayes90]	CON	Barry Hayes. Open Systems Require Conservative Garbage Collection. In Workshop: Garbage Collection in Object-Oriented Systems, 1990. Available at ftp.diku.dk:/pub/GC90.
[Hayes91]	ALG	Barry Hayes. Using Key Object Opportunism to Collect Old Objects. In OOPSLA '91 Proceedings, SIGPLAN Notices, 26(11):33-46, 1991.
[Heymann91]	SUR	Jurgen Heymann. A Comprehensive Analytical Model for Garbage Collection. In SIGPLAN Notices, 26(8):50-59, 1991.
[HicCoh84]	PER	Tim Hickey und Jacques Cohen. Performance Analysis of On-the-Fly Garbage Collection. In Communications of the ACM, 27(11):1143-1154, 1984.
[Hölzle91]*		Urs Hölzle. The Myth of High Object Creation Rates. In Workshop: Garbage Collection in Object-Oriented Systems, 1991. Available at ftp.diku.dk:/pub/GC91.
[Hudson91]	FIN	Richard L. Hudson. Finalization in a Garbage Collected World. In Workshop: Garbage Collection in Object-Oriented Systems, 1991. Available at ftp.diku.dk:/pub/GC91.
[Johnson91a]	PAR PER	Douglas Johnson. The Case for a Read Barrier. In SIGPLAN Notices, 26(4):279-287, 1991.
[Johnson91b]	PAR PER	Douglas Johnson. Comparing Two Garbage Collectors. In Workshop: Garbage Collection in Object-Oriented Systems, 1991. Available at ftp.diku.dk:/pub/GC91.
[Juul90]	SUR	Niel Christian Juul. Workshop Report: Garbage Collection in Object-Oriented Systems. In OOPSLA/ECOOP '90 Addendum to the Proceedings, ?, ?(?):35-41, 1990.
[LanDup87]*	ALG	Bernard Lang und Francis Dupont. Incrementally Compacting Garbage Collection. In Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, SIGPLAN Notices, 22(7):253-263, 1987.

[Li90] RT Kai Li.
Real-Time Concurrent Collection in User Mode.
In Workshop: Garbage Collection in Object-Oriented Systems, 1990.
Available at ftp.diku.dk:/pub/GC90.

[Masotti91] C++ Glauco Masotti.
EC++: extended C++.
In JOOP: Journal of Object-Oriented Programming, ?(9):10-20, 1991.

[Moss91] GEN J. Eliot B. Moss.
The UMass Language Independent Garabge Collector Toolkit.
In Workshop: Garbage Collection in Object-Oriented Systems, 1991.
Available at ftp.diku.dk:/pub/GC91.

[NetO'To93] RT Scott Nettles und James O'Toole.
Real-Time Replication Garbage Collection.
In Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 28(6):217-226, 1993

[Nilsen91] RT Kelvin Nilsen.
A High-Performance Architecture for Real-Time Garbage Collection.
In Workshop: Garbage Collection in Object-Oriented Systems, 1991.
Available at ftp.diku.dk:/pub/GC91.

[PlaSha91] DIS David Plainfossé und Marc Shapiro.
A Distributed Garbage Collection as an Operating System Component.
In Workshop: Garbage Collection in Object-Oriented Systems, 1991.
Available at ftp.diku.dk:/pub/GC91.
net: dp@sor.inria.fr

[QueBeaQue89] PAR Christian Queinnec, Barbara Beaudoing und Jean-Pierre Queille.
Mark DURING Sweep rather than Mark THEN Sweep.
In PARLE'89. Parallel Architectures and Languages Europe, Band 1, Seiten 224-237, 1989.

[Sankaran94] SUR Nandakumar Sankaran.
A Bibliography on Garbage Collection.
Technischer Report, Clemson University, South Carolina 29631, 1994.

[SchWai67]* H. Schorr und W. Waite.
An efficient machine-independent procedure for garbage collection in various list structures.
In Communications of the ACM, 10(8):501-506, 1967.

[ShaSof91] PAR GEN Ravi Sharma und Mary Lou Soffa.
Parallel Generational Garbage Collection.
In OOPSLA '91 Proceedings, SIGPLAN Notices, 26(11):16-32, 1991.

[Stroustrup91]* C++ Bjarne Stroustrup.
The C++ Programming Language.
Addison Wesley Publishing Company, 2. Auflage, 1991.

[Ungar84]* GEN David Ungar.
Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm.
In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Seiten 157-167, 1984.

[UngJac91] INC David Ungar und Frank Jackson.
Outwitting GC Devils: A Hybrid Incremental Garbage Collector.
In Workshop: Garbage Collection in Object-Oriented Systems, 1991.
Available at ftp.diku.dk:/pub/GC91.
net: david.ungar@sun.com, jackson@parcplace.com

[Vogt91] SUR Dr. Carsten Vogt.
Speicherorganisation in objektorientierten Systemen - eine Ubersicht.
In Informationstechnik, 33(4):208-219, 1991.

[Wentworth90]* CON E. P. Wentworth.
Pitfalls of Conservative Garbage Collection.
In Software - Practice and Experience, 20(7):719-727, 1990.

[WilLamMoh 91] ALG Paul R. Wilson, Michael S. Lam und Thomas G. Moher.
Effective 'Static-graph' Reorganisation to Improve Locality in Garbage-Collected Systems.
In Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 26(6):177-191, 1991.

[WilMoh89] Paul R. Wilson und Thomas G. Moher.
Design of the Opportunistic Garbage Collector.
In OOPSLA '89 Proceedings, SIGPLAN Notices, 24(10):23-35, 1989.

[Wilson88]* ALG Paul R. Wilson.
Opportunistic Garbage Collection.
In SIGPLAN Notices, 23(12):98-102, 1988.

[Wilson94]* SUR Paul R. Wilson.
Uniprocessor Garbage Collection Techniques.
Draft, submitted to ACM Computing Surveys, 1994
Available at cs.utexas.edu:/pub/garbage.

[Withington91] RT P. T. Withington.
How Real is "Real-Time" GC?
In Workshop: Garbage Collection in Object-Oriented Systems, 1991.
Available at ftp.diku.dk:/pub/GC91.
net: ptw@jasper.scrc.symbolics.com

[Zorn90]* CMP Benjamin Zorn.
Comparing Mark-and-sweep and Stop-and-copy Garbage Collection.
In Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, Seiten 87-98, 1990.

C Legende