Garbage-Collection
Grundlagen und Implementierungen in C++

Stefan Kuhlins, Karsten Meinders und Martin Schader
Universität Mannheim
Schloß
D-68131 Mannheim

29. Juni 1994

Zusammenfassung

Nach der Motivation, warum *Garbage-Collection* (GC) für das Programmieren mit dynamisch angelegten Objekten wünschenswert ist, werden die wichtigsten Verfahren erläutert. Anschließend stellen wir unsere Implementierungen der GC in C++ vor. Im Ausblick werden Entwicklungspotentiale der GC in C++ aufgezeigt.

Fehlerrisiken bei der expliziten Objektfreigabe

In vielen Bereichen der Software-Entwicklung werden komplexe Datenstrukturen mittels Zeigern auf Objekte, die auf dem Heap angelegt werden, realisiert. In der objektorientierten Entwicklung verwendet man beispielsweise Konstrukte wie Container-Klassen oder Aggregationsstrukturen. Probleme bereitet hier die korrekte Freigabe nicht mehr benötigter Strukturen, die in der Sprache C++ explizit vorgenommen werden muß.

Verlorene Objekte

Verlorene Objekte werden von keinem Zeiger referenziert. Ein solches Objekt entsteht dadurch, daß der letzte es referenzierende Zeiger gelöscht oder umgesetzt wird, ohne das Objekt freizugeben (vgl. Abb. 1). Der von einem verlorenen Objekt belegte Speicherplatz ist dann nicht mehr nutzbar. Es tritt ein *Speicherleck* auf, das u. U. zu knapp werdendem Speicher führt.

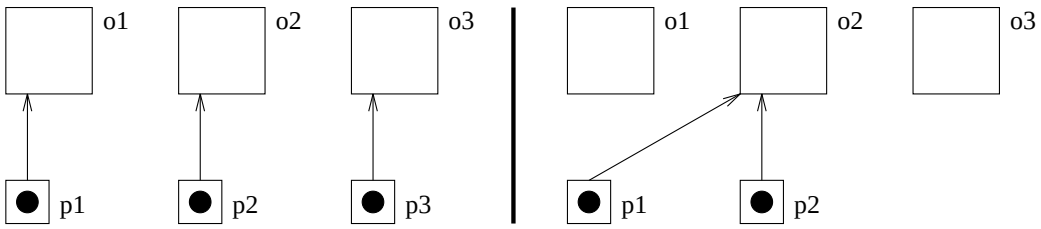

Abbildung 1: Verlorene Objekte `o1` und `o3` nach dem Umsetzen des Zeigers `p1` bzw. Löschen von `p3`.

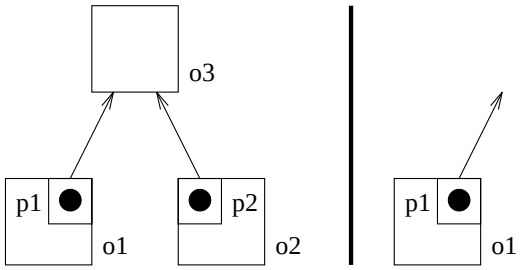

Abbildung 2: Hängender Zeiger `p1` nach dem Löschen der Objekte `o2` und `o3`.

Hängende Zeiger

Wird ein Objekt freigegeben, obwohl noch mindestens ein Zeiger auf das Objekt verweist, wird dieser zum *hängenden Zeiger*. Sofern über einen hängenden Zeiger Zugriffe vorgenommen werden, kann dies zu schwerwiegenden Fehlern führen, wenn der entsprechende Speicherplatz bereits anderweitig genutzt wird.

Hängende Zeiger treten typischerweise bei der Verwendung von *Eltern-Kind-Strukturen* auf. In Abb. 2 ist dargestellt, wie `p1` zum hängenden Zeiger in dem Elternobjekt `o1` wird, weil das Elternobjekt `o2` bei seiner Löschung das Kindobjekt `o3` freigibt.

Garbage-Collection als Ausweg

GC umfaßt Verfahren, die das Entstehen der beiden genannten Probleme verhindern sollen. Programmierer müssen diese nicht mehr explizit berücksichtigen und können sich somit auf die eigentliche Problemlösung konzentrieren.

Der Haupteinwand gegen die Verwendung von GC ist, daß sie in der Regel langsamer ist als die explizite Objektfreigabe. Hier muß zwischen Sicherheit und Komfort auf der einen sowie Geschwindigkeit auf der anderen Seite abgewogen werden. Moderne GC-Verfahren relativieren jedoch die Geschwindigkeitseinbußen, und darüber hinaus gibt es Situationen, in denen GC immer überlegen ist, beispielsweise bei den im Zusammenhang mit Eltern-Kind-Strukturen auftretenden Problemen.

Garbage-Collection-Verfahren

Alle GC-Verfahren sind auf die Kooperation mit dem jeweiligen Anwendungsprogramm angewiesen. Sie unterscheiden sich im Grad der benötigten Kooperation und in der Art und Weise, wie sie versuchen, den Laufzeitaufwand zu reduzieren.

Referenzenzählen

Das *Referenzenzählen* [COL60] gehört zu den ältesten Verfahren. Für jedes angelegte Objekt wird ein Zähler unterhalten, der die Anzahl der Referenzen auf dieses Objekt verwaltet. Wenn einem Zeiger die Adresse eines Objekts zugewiesen wird, so wird der Wert des Zählers für dieses Objekt um Eins inkrementiert. Sofern der Zeiger vorher auf ein anderes Objekt verwiesen hat, wird dessen Zähler entsprechend um Eins dekrementiert. Ebenso wird dekrementiert, wenn ein Zeiger gelöscht wird. Erreicht der Wert eines Referenzenzählers Null, kann der Speicherplatz des zugehörigen Objekts wiederverwendet werden, da es sonst ein verlorenes Objekt wird.

Das Referenzenzählen ist ein *unterbrechungsfreies* Verfahren, d. h. ein Programm muß nicht für einen „Durchlauf“ der GC angehalten werden. Der Aufwand hängt davon ab, wie oft Referenzen auf ein Objekt gebildet und wieder entfernt werden. Denn für jede Initialisierung, jede Zuweisung und jedes Löschen eines Zeigers muß der Referenzenzähler aktualisiert werden. In Anwendungen, die intensiven Gebrauch von Zeigern machen, kann daher die Laufzeit spürbar beeinträchtigt werden.

Ein grundsätzliches Problem beim Referenzenzählen sind *Zyklen*. Ein Zyklus liegt vor, wenn sich zwei oder mehr Objekte gegenseitig referenzieren (vgl. Abb. 3). Dadurch haben die Zähler der beteiligten Objekte zumindest einen Wert von Eins, und die Objekte können nicht mehr freigegeben werden.

Mark-and-Sweep Garbage-Collection

Das *Mark-and-Sweep-Verfahren* [MCC60] ist ebenso wie das Referenzenzählen ein GC-Klassiker. Um festzustellen, welche Objekte referenziert werden, macht sich das Verfahren zunutze, daß Heap-Objekte mit ihren Zeigern einen Graphen bilden. Dabei ist wesentlich, ob ein Objekt direkt oder indirekt über andere Zeiger von einem *Wurzelzeiger* erreichbar ist. Wurzelzeiger sind Zeiger, die sich im statischen Datenbereich, auf dem Stack oder in Registern – also nicht ebenfalls auf dem Heap – befinden.

Ausgehend von den Wurzelzeigern werden zunächst alle erreichbaren Objekte markiert (vgl. Abb. 4). In einem zweiten Durchlauf wird dann der Speicherplatz der nicht markierten Objekte freigegeben, und die Markierungen der erreichbaren Objekte werden wieder entfernt (vgl. Abb. 5). Entscheidend ist hier, daß die

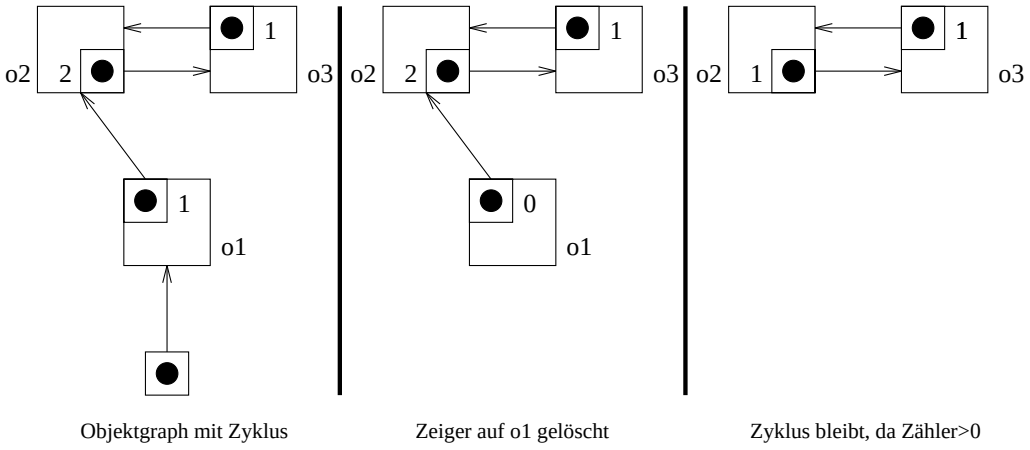

Abbildung 3: Zyklus-Problem beim Referenzenzählen

Adressen der Wurzelzeiger und der Zeiger in den Heap-Objekten zur Laufzeit bekannt sein müssen.

Kopierende Garbage-Collection

Vom Mark-and-Sweep-Verfahren wurde die *kopierende GC* [FY69] abgeleitet. Statt sie zu markieren, werden hier alle erreichbaren Objekte innerhalb des Heaps umkopiert, so daß an einem Ende des Heaps ein lückenloser freier Speicherbereich entsteht. Somit entfällt der zweite Durchlauf. Der Laufzeitaufwand hängt nur von der Anzahl der erreichbaren Objekte und nicht von der Anzahl aller existierender Objekte ab.

Mehrfachreferenzen auf ein Objekt erfordern bei der kopierenden GC Kooperation mit dem Anwendungsprogramm in Form von *indirekten* oder *weiterleitenden Zeigern*. Ein indirekter Zeiger verweist auf ein Element einer Objekttabelle, die die Objektadressen speichert (vgl. Abb. 6). Falls ein Objekt umkopiert wird, muß somit nur seine Adresse in der Objekttabelle modifiziert werden. Im zweiten Fall wird ein weiterleitender Zeiger, der die neue Objektadresse enthält, an die Stelle des umkopierten Objekts gesetzt (vgl. Abb. 7). Beim Dereferenzieren wird über ihn auf das umkopierte Objekt zugegriffen.

Parallele und inkrementelle Garbage-Collection

Durch die Verfügbarkeit parallel arbeitender Prozessoren motiviert, richteten sich die Anstrengungen bei der Weiterentwicklung der Verfahren auf *parallele* und *inkrementelle GC*. Parallele GC läuft zeitgleich mit dem Anwendungsprogramm, während inkrementelle GC sich mit seiner Bearbeitung abwechselt. In beiden

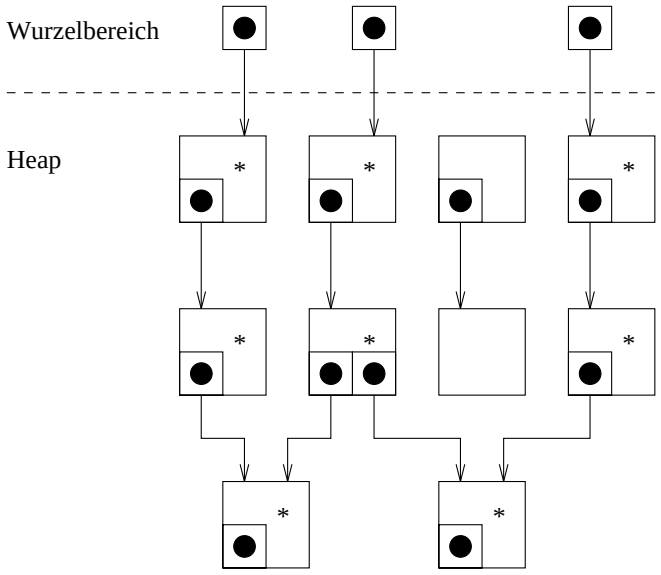

Abbildung 4: Objektgraph nach dem ersten Durchlauf

Fällen ist eine anspruchsvolle Kooperation mit dem Anwendungsprogramm erforderlich, um die Konsistenz der Referenzen sicherzustellen. Arbeiten in dieser Richtung konzentrieren sich daher mehr auf Korrektheitsbeweise [DLM78] und weniger auf die praktische Implementierung. Sie sollen deshalb hier nicht weiter vertieft werden.

Generationenbasierte Garbage-Collection

Viele neu angelegte Objekte werden bereits nach kurzer Zeit nicht mehr benötigt. Ein länger lebendes Objekt wird dagegen oft auch zukünftig benötigt. Diese Beobachtung macht sich die *generationenbasierte GC* [UNG84] zunutze, bei der die Objekte in eine *junge* und in eine *alte Generation* unterteilt werden. Die alte Generation wird grundsätzlich als überlebend betrachtet, so daß bei einem GC-Durchlauf lediglich die junge Generation zu untersuchen ist, wodurch sich drastische Laufzeiteinsparungen ergeben können. Junge, überlebende Objekte werden in die alte Generation übernommen. Eine *Ausnahmetabelle* speichert die Adressen derjenigen jungen Objekte, die von alten Objekten referenziert werden, um das irrtümliche Freigeben dieser jungen Objekte zu verhindern (vgl. Abb. 8). Dazu müssen alle Zeigerzuweisungen überwacht werden. In größeren Zeitabständen müssen darüber hinaus alle existierenden Objekte durchsucht werden, um alte Objekte zu finden, die nicht mehr gebraucht werden.

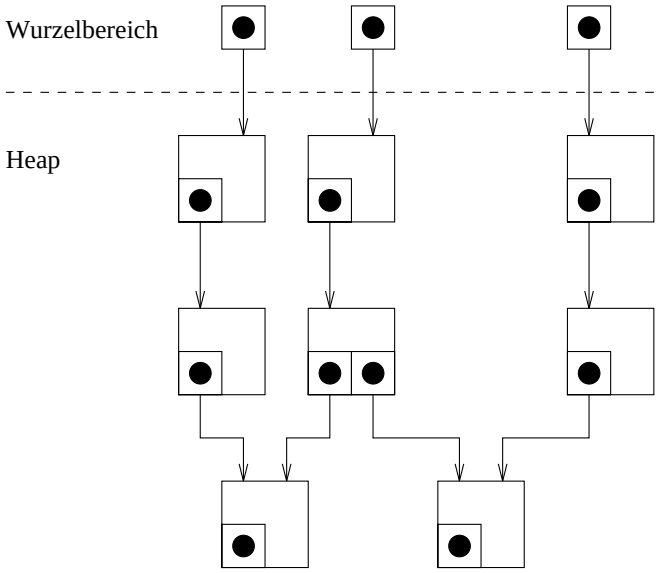

Abbildung 5: Objektgraph nach dem zweiten Durchlauf

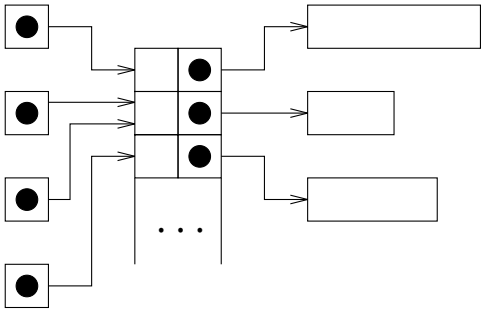

Abbildung 6: Indirekte Zeiger und Objekttabelle

Konservative Garbage-Collection

Konservative (nicht kopierende) GC [BW88] basiert auf dem Mark-and-Sweep-Verfahren, wobei die Zeigeradressen nicht explizit bekannt sein müssen, sondern gesucht werden (sogenanntes *Referenzensuchen*). Deshalb ist es das einzige Verfahren, das ohne Modifikation der Anwendungsprogramme oder Compiler auskommt.

Jedes Wort im *Wurzelbereich* (der die Wurzelzeiger enthält) und in existierenden Heap-Objekten, das einer gültigen Objektadresse entsprechen könnte, wird als Zeiger interpretiert. Über solche „Zeiger“ gefundene Objekte werden markiert. Die Zeigerverifizierung wird beispielsweise dadurch unterstützt, daß alle Heap-Objekte nach bestimmten Adressen ausgerichtet werden, so daß „Zeiger“, die nicht auf solche Adressen verweisen, von vornherein ausgeschlossen werden können.

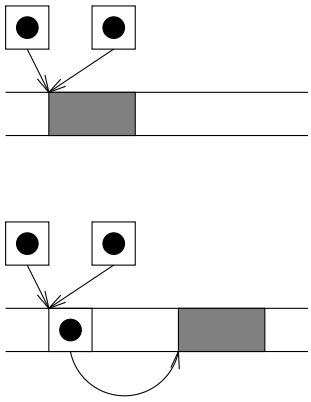

Abbildung 7: Weiterleitender Zeiger

Ein beliebiges Speicherwort kann hier zufällig der Adresse eines verlorenen Objekts entsprechen (*Zeigerfehlidentifizierung*). Ein Lösungsvorschlag für dieses Problem ist das sogenannte *Blacklisting* [BOE93], bei dem vor der ersten Speicherplatzzuordnung kritische Heap-Bereiche gesperrt werden.

Implementierungen in C++

Wir haben verschiedene Ansätze für das Referenzensuchen und das Referenzenzählen in C++ implementiert. Ihre Vor- und Nachteile werden im folgenden diskutiert. Die Quelltexte unserer Implementierungen sind per *Anonymous-ftp* auf dem Server der Universität Mannheim (`ftp.uni-mannheim.de`) im Unterverzeichnis `/pub/gc` zugänglich.

Referenzenzählen in C++

Das Referenzenzählen wurde unter dem Namen MARC (*Memory Administration with Reference Counting*) entwickelt. Wie oben erläutert, basiert es auf der Aktualisierung der Referenzenzähler bei allen Zeigeroperationen. Hierzu wurden parametrisierte Zeigerklassen mit geeigneten Konstruktoren, Destruktor und dem überladenen Zuweisungsoperator definiert. Zur Vervollständigung der Klassen wurden außerdem die Operatoren `!`, `[]`, `->` sowie die Adreßarithmetikoperatoren wie beispielsweise `++` überladen.

Um für jedes angelegte Objekt einen Referenzenzähler zu erzeugen, muß hier der Operator `new` global überladen werden. Die eigene Speicherverwaltung kopiert Objekte um, damit der Speicher möglichst unfragmentiert bleibt. Dies ist möglich, weil die Zeigerklassen als indirekte Zeiger konzipiert sind. Sobald der Zählerwert eines Objekts Null erreicht, wird der Destruktor des Objekts explizit aufgerufen, so daß *Finalization Code*, wie beispielsweise das Schreiben eines Puffers in eine

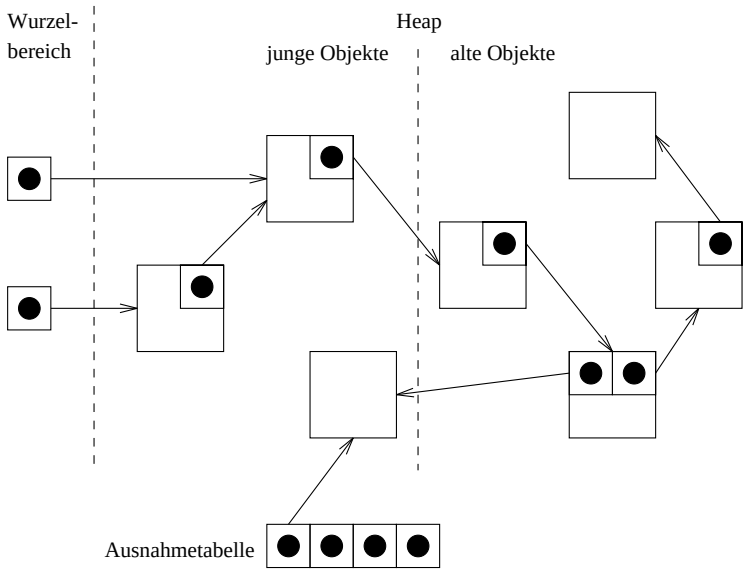

Abbildung 8: Generationenbasierte GC mit Ausnahmetabelle

Datei, ausgeführt werden kann.

Der Einsatz von MARC setzt voraus, daß die Header-Datei `marc.h` eingebunden wird. Weiterhin sind in allen Zeigerdeklarationen spezielle Zeigerklassen einzusetzen. Beispielsweise muß

```
class X {
   X* p;
   // ...
};
X* xp = new X;
```

umgeschrieben werden in

```
class X {
   ClassPtr<X> p;
   // ...
};
ClassPtr<X> xp = new X;
```

Die Absicht, das Referenzenzählen nicht nur prototypisch für die Zeiger einer Klasse zu entwickeln, sondern allgemeingültig einsetzbar zu machen, führt zu Problemen, weil Zeigeroperationen nicht typunabhängig sind. So steht für den Typ `void*` keine Zeigerarithmetik zur Verfügung, und der Pfeiloperator `->` ist ausschließlich für Zeiger auf Klassenobjekte definiert. Somit ist eine Hierarchie von drei Zeigerklassen notwendig. Die Basisklasse `VoidPtr` ersetzt Zeiger des Typs

`void*`. Die davon abgeleitete, parametrisierte Klasse `StdPtr<T>` ersetzt Zeiger auf vordefinierte Datentypen, und die wiederum von `StdPtr<T>` abgeleitete Klasse `ClassPtr<T>` ersetzt ausschließlich Zeiger auf Klassenobjekte, z. B.

```
VoidPtr       vp;             // richtig
StdPtr<int>   ip1 = new int;  // richtig
ClassPtr<int> ip2 = new int;  // falsch
StdPtr<X>     xp1 = new X;    // falsch
ClassPtr<X>   xp2 = new X;    // richtig
```

Beachtet werden muß auch der Fall, daß ein Zeigerklassenobjekt auf ein Objekt zeigt, das sich nicht auf dem Heap befindet:

```
X x;                   // automatische Variable,
ClassPtr<X> xp = &x;   // kein Heap-Objekt
```

Dies ist problematisch, weil jetzt kein Referenzenzähler für ein solches Objekt bereitsteht. In einem solchen Fall wird automatisch ein Hilfszugriffselement eingeführt, so daß `StdPtr<T>` und `ClassPtr<T>` auch für derartige Objekte verwendet werden können. Dies ist insbesondere für Initialisierungen mit dem Nullzeiger wichtig.

```
ClassPtr<X> xp = 0;  // Nullzeiger
```

Insgesamt wurde die Implementierung erheblich umfangreicher, komplexer und schwerfälliger als anfangs angenommen.

Referenzensuchen in C++

Das Referenzensuchen wurde in verschiedenen Varianten implementiert, die im Gegensatz zu MARC alle nicht portabel sind. Denn die Lage des Stacks und der statischen Datenbereiche, die Anzahl der Register, die Größe der Zeiger und die Ausrichtung der Zeiger und der Objekte sind maschinenabhängig. (Unsere Implementierungen können mit Borland C++ auf PC unter DOS übersetzt und ausgeführt werden.)

Referenzensuchen mit der Speicherverwaltung Quick-Fit

Dieses Verfahren wurde unter dem Namen MARS (*Memory Administration with Reference Searching*) implementiert. MARS ist einfach anzuwenden. Außer der Einbindung der Header-Datei `mars.h` muß im Quelltext nichts geändert werden.

Bei der Speicherverwaltung *Quick-Fit* enthält ein *Größenfeld* Listen mit jeweils gleich großen freien Speicherbereichen. Mit der Angabe der Größe eines neu anzulegenden Objekts als Index erhält man einen freien Bereich (vgl. Abb. 9).

Ist eine Liste leer, wird ein neuer *Pool* für Objekte der gewünschten Größe auf dem Heap angelegt. Freie Fragmente jenseits einer Maximalgröße werden in einer gewöhnlichen Freiliste verwaltet. Kleine Objekte werden hier im Vergleich zum Standardoperator `new` wesentlich schneller angelegt.

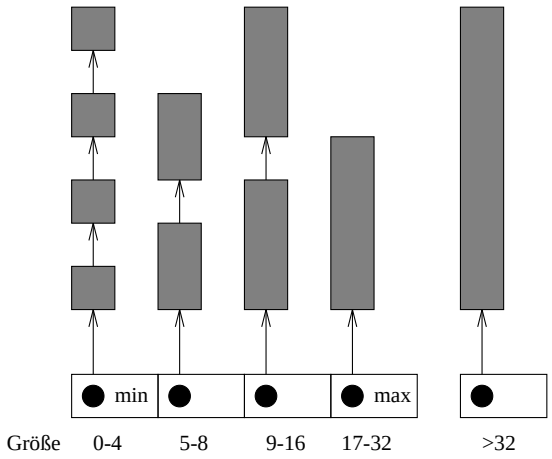

Abbildung 9: Speicherverwaltung Quick-Fit

Die Überprüfung potentieller Zeiger muß sehr schnell durchführbar sein. Dadurch, daß die Pools und auch die Objekte innerhalb der Pools nach festen Adressen ausgerichtet sind, kann die Zeigerverifizierung allein durch einfache Bitoperationen bewerkstelligt werden. Im ungünstigsten Fall braucht ein GC-Durchlauf so nicht mehr Zeit als die explizite Freigabe der entsprechenden Objekte. Jedoch können Destruktoren nicht aufgerufen werden, weil der Typ eines Objekts bei der Adreßüberprüfung nicht bekannt ist.

Bei Laufzeittests traten viele Zeigerfehlidentifizierungen auf, weil Speicherworte im statischen Datenbereich gültigen Objektadressen entsprachen. Das Blacklisting brachte in unseren Tests keine Abhilfe.

Präzise konservative Garbage-Collection

Bei der *präzisen GC* sind die Adressen der Zeiger im Speicher bekannt, wodurch Zeigerfehlidentifizierungen ausgeschlossen werden. In der Implementierung tragen Zeiger bei ihrer Erzeugung ihre Adresse in eine *Lokationsliste* ein. Wie bei MARC werden drei Zeigerklassen benötigt. Im Konstruktor einer Zeigerklasse wird die Adresse eines Stack- oder eines statischen Zeigerklassenobjekts in die Liste eingetragen. Für Zeiger in Heap-Objekten werden in einer Initialisierungsphase statische Klassenzeigerlisten erzeugt.

Die Heap-Objekte werden wie beim Mark-and-Sweep-Verfahren mit einem Graphendurchlaufverfahren gesucht und markiert. Da die Zeiger der Zeigerklassen über Typ-Informationen verfügen, kann die benötigte Klassenzeigerliste ge-

funden werden. Die GC bleibt konservativ, weil auf das Umkopieren von Objekten verzichtet wurde.

Ein GC-Durchlauf bei großen Objekten ist schneller als beim oben geschilderten Referenzensuchen, da die in einem Objekt enthaltenen Zeiger nicht gesucht werden müssen, sondern bekannt sind. Allerdings dauert die Initialisierung eines Zeigers vom Typ `ClassPtr<X>` verglichen mit einem Standardzeiger `X*` länger. Dies wirkt sich auf die Gesamtdauer der Speicherplatzzuweisungen aus. Wie bei MARC muß im Quelltext jede Zeigervariablendeklaration durch eine Zeigerklassendeklaration ersetzt werden.

Entwicklungspotentiale

Das präzise Referenzensuchen leidet darunter, daß die Lokationslisten zur Laufzeit gebildet werden müssen. Stünden sie bereits zur Verfügung, ergäbe sich kein Zusatzaufwand. Daher wird hier die mögliche Integration der GC in C++-Compiler einschließlich weiterer Verbesserungsmöglichkeiten diskutiert.

Schlüsselobjekte

Untersuchungen in [HAY91] haben gezeigt, daß langlebige Objekte oft in *Clustern* auftreten, die zum gleichen Zeitpunkt inaktiv werden. Solche Cluster bestehen aus Objekten, die einen Baum oder ähnliche Datenstrukturen bilden. Definiert man für ein Cluster *Schlüsselobjekte*, die bei allen GC-Durchläufen überprüft werden, so ist das ganze Cluster nur zu durchsuchen, wenn die zugehörigen Schlüsselobjekte nicht markiert wurden.

Vorhersage der Lebenszeit

Die *Vorhersage der Lebenszeit* wird als Weg zu weiterer Laufzeitverbesserung gesehen [BZ93]. Objekte mit ähnlicher Lebenszeit sollten räumlich nahe beieinander angelegt werden. Dies kann bei virtuellem Speichermanagement das Nachladen von Seiten positiv beeinflussen, weil Seiten nicht mit inaktiven Objekten durchsetzt sind. Die Zeitpunkte, an denen ein Cluster inaktiv wird, sind ebenfalls aus den Lebenszeitvorhersagen abzuleiten. Ein GC-Durchlauf zu einem solchen Zeitpunkt hat den Vorteil, daß erwartungsgemäß sehr viele Objekte inaktiv sind.

Hier besteht das Problem darin, ein geeignetes Verfahren für die Bestimmung der Lebenszeiten anzugeben. Unterschieden wird zwischen statischen und dynamischen Verfahren. Statische Verfahren benutzen Informationen der Quelltextanalyse, während dynamische Verfahren den Zustand der CPU als Prognosemittel verwenden. Statische Informationen sind beispielsweise der Typ der Objekte oder die Funktion, in der ein Objekt angelegt wird. Für C++ würde sich auch die Lebenszeit der Zeiger eignen, die auf ein Objekt verweisen. Die Lebenszeit

des langlebigsten Zeigers, beispielsweise eines statischen Zeigers, bestimmt die maximale Lebenszeit des Objekts. Zur Laufzeit entsteht kein Zusatzaufwand im Gegensatz zu den dynamischen Verfahren.

Compiler-Integration

Um eine präzise GC zu ermöglichen, müssen Lokationslisten für Zeiger auf dem Stack, in Objekten und im statischen Datenbereich angelegt werden. Für die Verwaltung der Generationen müssen Zeiger in alten Klassenobjekten daraufhin überwacht werden, ob ihnen die Adresse von jungen Objekten zugewiesen wird. In solchen Fällen werden sie in die Ausnahmetabelle aufgenommen. Dies wirkt sich auf die Laufzeit nur unwesentlich aus, weil nur Zeiger in Heap-Objekten der alten Generation betroffen sind, für die der entsprechende Code erzeugt werden muß.

Durch die Compiler-Integration der GC ließen sich die Nachteile, die durch eine eigenständige GC-Implementierung entstehen, vollständig vermeiden, ohne daß der Quelltext bestehender Programme geändert werden muß.

Folgt man jedoch der klassischen C++-Philosophie, daß Anwender nur für das „bezahlen“ sollen, was sie auch benutzen, sollte GC zwar in die Sprache C++ integriert werden, aber optional ein- und abschaltbar sein – z. B. auf Klassenebene [STR94].

Literatur

[BZ93] D. A. Barrett, B. G. Zorn: *Using Lifetime Prediction to Improve Memory Allocation Performance*, Proceedings of the ACM SIGPLAN '93 Conference on Programming Languages Design and Implementation, 1993, S. 187–196.

[BOE93] H.-J. Boehm: *Space Efficient Conservative Garbage Collection*, Proceedings of the ACM SIGPLAN '93 Conference on Programming Languages Design and Implementation, 1993, S. 197–206.

[BW88] H.-J. Boehm, M. Weiser: *Garbage Collection in an Uncooperative Environment*, Software-Practice and Experience, Vol 18(9), 1988, S. 807–820.

[COL60] G. E. Collins: *A Method for Overlapping and Erase of Lists*, Communications of the ACM, Vol. 3(12), 1960, S. 503–507.

[DLM78] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, E. F. M. Steffens: *On-the-Fly Garbage Collection: An Exercise in Cooperation*, Communications of the ACM, Vol. 21(11), 1978, S. 966–975.

[FY69] R. R. Fenichel, J. C. Yochelson: *A LISP garbage collector for virtual-memory computer systems*, Communications of the ACM, Vol. 12(11), 1969, S. 611–612.

[HAY91] B. Hayes: *Using Key Object Opportunism to Collect Old Objects*, ACM OOPSLA '91, 1991, S. 33–46.

[MCC60] J. McCarthy: *Recursive Functions of Symbolic Expressions and Their Computation by Machine*, Part I, Communications of the ACM, Vol. 3(4), 1960, S. 184–195.

[STR94] B. Stroustrup: *The Design and Evolution of C++*, Addison-Wesley, 1994, S. 219–222.

[UNG84] D. Ungar: *Generational Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm*, ACM SIGPLAN notices, Vol. 19(5), 1984, S. 157–167.