Spirick Tuning

Die C++ Klassen- und Template-Bibliothek

für performancekritische Anwendungen

Tutorial

Zur Benutzung des Tutorials

Das vorliegende Werk beschreibt grundlegende Konzepte und Techniken der Bibliothek **Spirick Tuning**. Um die Darstellung möglichst allgemeinverständlich zu halten, wurden gegenüber dem Originalquelltext zahlreiche Vereinfachungen vorgenommen. Es kann daher vorkommen, daß Begriffe, Namen von Datentypen, Klassen, Funktionen und Methoden und Beziehungen zwischen Klassen, die im Tutorial beschrieben werden, vom Originalquelltext abweichen. Eine umfassende und gültige Beschreibung des Produkts **Spirick Tuning** befindet sich ausschließlich im zugehörigen Referenzhandbuch.

Inhaltsverzeichnis

1 Performance-Analyse eines C++-Programms

1.1 Einleitung

In den vergangenen Jahren hat sich die Softwareentwicklung grundlegend gewandelt. Objektorientierte Techniken und Werkzeuge verließen die Labors, in denen sie jahrelang gereift waren. Die neue Programmiersprache C++ entstand. Sie erweiterte die anerkannte Sprache C um objektorientierte Konzepte, entwickelte sich bald zu einem De-Facto-Industriestandard und trug wesentlich zur heutigen Verbreitung der Objektorientierung bei.

Während des Übergangs von der strukturierten zur objektorientierten Programmierung wurde nicht einfach die Programmiersprache C durch C++ ersetzt. Es fand ein grundlegender Wandel in allen Phasen der Programmentwicklung statt. Neben der Implementierung wurden auch Analyse und Design neu gestaltet. Die objektorientierte Vorgehensweise erhöht wesentlich die Übersichtlichkeit. Damit verringert sich die Häufigkeit von Fehlern, und der Aufwand für Entwicklung und Wartung eines Programms sinkt enorm.

Ein objektorientiertes Programm erhält schon in der Entstehungsphase ein anderes Gesicht. Neue Konzepte wie Datenkapselung und Polymorphie finden Einzug in Analyse und Design. Bei der Implementierung werden mehr und kleinere Objekte eingesetzt. Statt globaler Funktionen werden Methoden der Objekte aufgerufen. Für das Erzeugen und Löschen der Objekte werden Konstruktoren und Destruktoren verwendet. Sie sorgen für die Konsistenz der verwalteten Daten. Mit virtuellen Methoden wird es möglich, Objekte zu verwenden, deren genauer Typ nicht bekannt ist.

Die heutigen objektorientierten Programmiersprachen besitzen aber noch Mängel in bezug auf die Laufzeiteffizienz. Die Umstellung eines Projekts auf objektorientierte Techniken ist oft mit einem erhöhten Bedarf an Ressourcen verbunden. Dieser Performanceverlust ist jedoch keine unheilbare Krankheit. Im vorliegenden Buch werden die Schwachstellen eines C++-Programms aufgedeckt und behandelt. Dabei zeigt sich, daß Objektorientierung und High Performance keine Gegensätze sind. Alle vorgeschlagenen Konzepte passen sich harmonisch in die objektorientierte Sprache C++ ein.

Einige Leser werden sich fragen, ob sich der Aufwand für das Performance-Tuning wirklich lohnt. Schließlich gelangt mit rasantem Tempo neue, leistungsfähigere Hardware auf den Markt. Auch die Programmierwerkzeuge werden ständig verbessert. Dabei wird aber übersehen, daß unsere Ansprüche schneller wachsen als die Hardware. Noch vor wenigen Jahren benötigten wir für das Verarbeiten großer Datenmengen einen "großen" Computer. Heute werden statt Mainframes zunehmend Rechnernetze eingesetzt. Damit steigen die Ansprüche an die Rechenleistung unseres persönlichen Arbeitsplatzcomputers. Bei der Auswahl geeigneter Software spielt die Performance eine wichtige Rolle. Beschäftigt sich ein Programm vorwiegend mit sich selbst, statt unsere Aufgaben zu lösen, sehen wir uns nach einem anderen um.

C++ verfügt über bessere Optimierungsmöglichkeiten als andere objektorientierte Sprachen. Diese Möglichkeiten sind aber nicht offensichtlich, und sie werden vom Compiler nicht automatisch eingesetzt. Kennen wir sie nicht, ist die Performance unseres Programms

nur durchschnittlich. Für eine bessere Performance müssen wir die *sprachnahen Werkzeuge* genauer untersuchen. Dazu zählen:

- Die Definition der Programmiersprache C++.
- Der C/C++-Compiler.
- Die C-Standardbibliothek mit den Modulen `string`, `stdlib` usw.

Wir betrachten diese Werkzeuge aus dem Blickwinkel der Performance. Das Buch liefert deshalb keinen vollständigen Überblick der Sprache C++. Stattdessen untersuchen wir diejenigen Teile, die den Ressourcenverbrauch unseres Programms beeinflussen (siehe Abbildung 1-1). Im Laufe der Untersuchungen werden wir mehrmals das Seziermesser an Compiler und Standardbibliothek anlegen. Die Ergebnisse sind nicht immer appetitlich. Wir erlangen aber die Fähigkeit, unser Programm von ungesundem Ballast zu befreien.

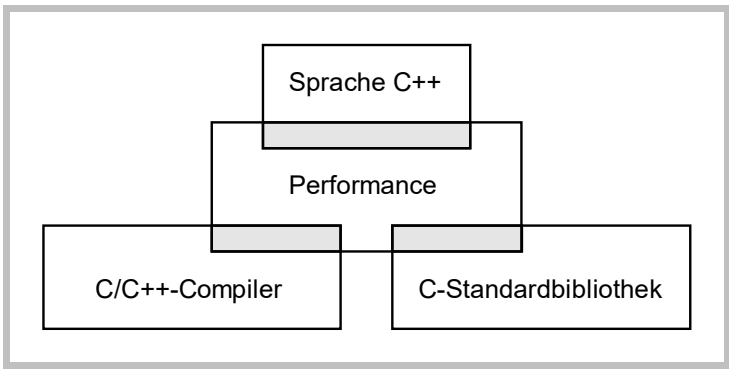

Abb. 1-1: *Grau: Der Gegenstand des Buchs*

Es existieren viele Wege, ein Programm zu optimieren. Auf die noch nicht bewiesene These, daß jedes Programm um mindestens ein Byte verkürzt werden kann, ohne an Funktionalität einzubüßen, gehen wir hier nicht ein. Stattdessen werden wir uns handfesten Tatsachen widmen. Das algorithmische Tuning ist seit vielen Jahren erforscht und in zahlreichen Büchern publiziert. Dazu zählen angepaßte Sortierverfahren, die Zugriffsbeschleunigung mit Hashtabellen usw. Je stärker wir ein Programm algorithmisch optimieren, desto höher werden unsere Ansprüche an die Sprachwerkzeuge (siehe Abbildung 1-2). Im Laufe des Buchs lernen wir zahlreiche Möglichkeiten kennen, auch sie optimal an unsere Bedürfnisse anzupassen.

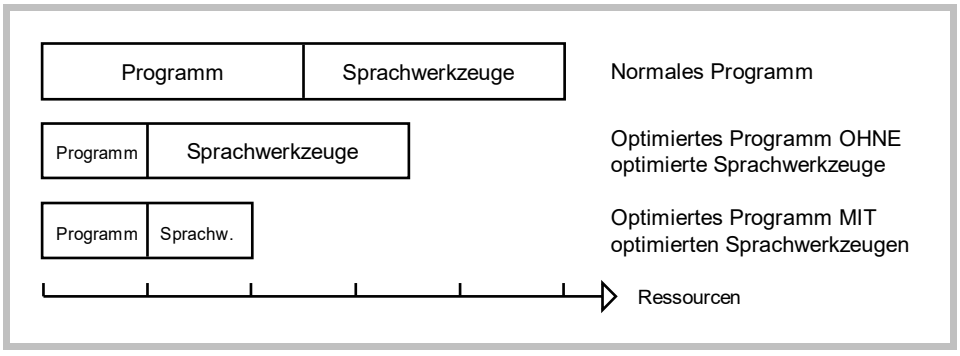

Abb. 1-2: *Anteile des Ressourcenverbrauchs*

Beim Optimieren eines Programms dürfen wir die *Qualität* nicht außer acht lassen. Wir werden beim Performance-Tuning weder unübersichtlichen Spaghetticode noch Assemblerroutinen einsetzen. Nach einem gründlichen Design finden wir auch für schwierige Probleme eine objektorientierte Lösung in C++. Das nachträgliche Ändern getesteter Programmteile ist stets mit Risiken verbunden. Deshalb sollten wir performancekritische

Stellen frühzeitig erkennen und einkapseln. Nützliches Zusatzwissen über die Spachwerkzeuge hilft uns dabei.

Optimierungskonzepte sind kein Gegensatz zu bestehenden, sondern eine sinnvolle Ergänzung. Wir können sie auch an nicht performancekritischen Stellen einsetzen. Manchmal stehen sie jedoch im Widerspruch zu anderen Design- und Programmierregeln. In diesen Fällen müssen wir sorgfältig abwägen, was die höhere Priorität besitzt. Moderne Programme mit graphischer Benutzeroberfläche bestehen aus zahlreichen Komponenten. Normalerweise sind davon nur zehn bis zwanzig Prozent performancekritisch. Diese beanspruchen siebzig bis achzig Prozent der Ressourcen. Da die performancekritischen Komponenten nur einen geringen Anteil am Gesamtprogramm besitzen, können wir dort auch abweichende Regeln einsetzen.

Das Performance-Tuning ist oftmals wie Goldsuche. Es existieren hunderte oder tausende Faktoren, die sich auf den Ressourcenverbrauch auswirken. Viele verstecken sich hinter Compilerschaltern und Standardfunktionen. Sind wir mit der Performance unseres Programms nicht zufrieden, greifen wir auf der Suche nach dem Flaschenhals oft ins Leere. Statt eines großen Klumpen Goldes finden wir meist nur einen kleinen Ressourcenkrümel. Deshalb Goldsucher und solche, die es werden wollen, aufgepaßt! Im vorliegenden Buch werden einige Gebiete gezeigt, in denen mit Sicherheit etwas zu finden ist.

Das Buch ist aus der Praxis entstanden und für Praktiker geschrieben. Es ist kein Lehrbuch und enthält keine Definitionen. Grundkenntnisse in der Programmiersprache C++ werden vorausgesetzt. Für Leser, die mit der objektorientierten Begriffswelt und C++ noch nicht vertraut sind, werden wichtige Begriffe in groben Zügen erläutert. Neue Begriffe werden nicht in einer Definition sondern im jeweiligen Kontext beschrieben.

Das Buch gliedert sich in drei große Kapitel. Diese Unterteilung entspricht den drei Phasen der Programmentwicklung: *Analyse*, *Design* und *Implementierung*. Im ersten Teil analysieren wir ein kleines C++-Programm und stellen fest, daß wir bei seiner Entwicklung in mehrere Performancefallen getappt sind. Der zweite Teil enthält Konzepte, mit denen wir die Performance verbessern können. Im dritten Teil werden sie in C++ umgesetzt.

1.2 Einige Grundlagen

1.2.1 Zur Notation im Buch

Im Verlauf des Buches werden wir uns zahlreiche Designdiagramme und Programmfragmente ansehen. Um deren Verständlichkeit zu erhöhen, sind sie alle nach einheitlichen Prinzipien erstellt. Für die objektorientierten Designdiagramme wird die **Methode von Peter Coad** verwendet. Abbildung 1-3 zeigt die wichtigsten graphischen Elemete dieser Designmethode.

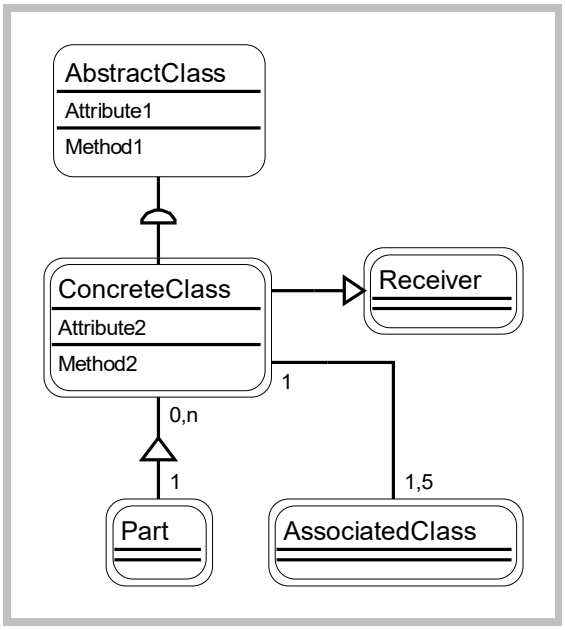

Abb. 1-3: *Graphische Elemente in Designdiagrammen*

Die Klasse `AbstractClass` ist eine abstrakte Basisklasse. Sie enthält rein virtuelle Methoden, und von ihr können keine Instanzen (Objekte) gebildet werden. Abstrakte Klassen sind einfach umrandet. Konkrete Klassen, zum Beispiel `ConcreteClass`, sind doppelt umrandet. Das umrandete Viereck einer Klasse besteht aus drei Teilen: Klassenname, Liste der Attribute und Liste der Methoden. Zwischen Klassen gibt es vier Arten von Verbindungen:

- Vererbung (*Inheritance*, *Generalization and Specialization*),
- Teil-Ganzes-Beziehung (*Whole Part Relation*),
- Objekt-Verbindung (*Object Connection*) und
- Nachrichten-Verbindung (*Message Connection*).

In Abbildung 1-3 erbt die Klasse `ConcreteClass` von `AbstractClass`. Die Klasse `Part` ist als Teil in der Klasse `ConcreteClass` enthalten. In einer Teil-Ganzes-Beziehung können Kardinalitäten angegeben werden. In unserem Beispiel bedeuten sie: Ein Objekt der Klasse `ConcreteClass` enthält null bis n Objekte der Klasse `Part`, und ein Objekt der Klasse `Part` gehört zu genau einem Objekt der Klasse `ConcreteClass`. Zwischen `ConcreteClass` und `AssociatedClass` besteht eine Objekt-Verbindung. Auch diese kann mit Kardinalitäten näher beschrieben werden. Die Bedeutung im Beispiel ist: Zu einem Objekt der Klasse `ConcreteClass` gibt es genau ein Objekt der Klasse `AssociatedClass`, und zu einem Objekt der Klasse `AssociatedClass` gibt es ein bis fünf Objekte der Klasse `ConcreteClass`.

Für Elemente der Programmiersprache C++ gelten die folgenden **Namenskonventionen**: Alle Namen sind aus englischen Wörtern oder deren Abkürzungen zusammengesetzt. Um ein Kauderwelsch im Programm zu vermeiden, werden keine deutschen Bezeichnungen verwendet. In C++ sind alle Schlüsselwörter englisch, zum Beispiel `class`, `public` oder `unsigned`. Außerdem unterstützt die Programmiersprache nicht die deutschen Umlaute. Sehen wir uns dazu ein Beispiel an.

```
Wenn (KannÖffnen (Datei)) .... // lesbar, aber kein C++
if (KannOeffnen (Datei)) ....  // C++, aber Kauderwelsch
if (CanOpen (File)) ....       // lesbar und C++
```

Bei zusammengesetzten Namen beginnt jedes Teilwort mit einem Großbuchstaben. Die Schreibung des ersten Buchstabens eines Namens ist von seinem Gültigkeitsbereich (*Scope*) abhängig. Namen, die global oder in einer Klasse gültig sind, beginnen mit einem Großbuchstaben. Lokale Namen, das heißt Parameter und lokale Variable einer Methode,

haben einen Kleinbuchstaben am Anfang. Alle Namen von Typen, Variablen und Attributen besitzen einen Präfix. Darin ist eine Kurzinformation über den Typ oder die Kategorie des Namens enthalten. Dadurch müssen wir nicht jedesmal nach der Definition des Namens suchen, wenn wir wissen möchten, wie er einzuordnen ist. Methodennamen wird kein Präfix vorangestellt. Dadurch würde sich die Lesbarkeit des Programmtextes erheblich verschlechtern. In Anlehnung an eine verbreitete C-Konvention schreiben wir Makros durchgängig mit Großbuchstaben. Einzelne Teilwörter werden durch einen Unterstrich voneinander getrennt. Der Präprozessor ist kein Bestandteil der Programmiersprache im engeren Sinn. Er analysiert den Text in einem separaten Paß vor dem eigentlichen C++-Compiler. Präprozessormakros sollten sich deutlich vom übrigen Programmtext abheben. Das folgende Programmfragment verdeutlicht mit einigen Beispielen die Bildung von Präfixen.

```
int           i_Number;               // Int
int *         pi_AddrOfNumber;        // Pointer to Int
char          c_Input;                // Character
char *        pc_AddrOfInput;         // Pointer to Character
char * *      ppc_AddrOfPtr;          // Pointer to pointer to Character
enum          et_Color {              // Enumeration Type
              ec_Blue }               // Enumeration Constant
              eo_BackgroundColor;     // Enumeration Object
typedef int   t_Counter;              // Type
t_Counter     o_Counter;              // Object
t_Counter *   po_AddrOfCounter;       // Pointer to Object
class         ct_TextPos;             // Class Type
ct_TextPos    co_TextPos;             // Class Object
ct_TextPos *  pco_AddrOfTextPos;      // Pointer to Class Object
template <class t> gct_ClassTemplate; // Generic Class Type
#define ASSERT(X) if (!(X)) Error (); // Preprocessor Macro
```

1.2.2 Überblick ist Alles

In der Designphase eines Programms entstehen wichtige Grundlagen. Klassen werden erzeugt und mit einem Interface versehen. Zwischen den Klassen werden Beziehungen hergestellt. Die Klassen mit ihren Vererbungsbeziehungen bilden einen **Klassenbaum**. Mit objektorientierten Werkzeugen ist es für den Designer einfach, einen Baum um weitere Klassen und Beziehungen zu erweitern. Wie gut verstehen aber spätere Anwender einen breit gefächerten Klassenbaum? Der Umgang mit einem Klassenbaum ist mit der Handhabung von standardisierten Kleingeräten, zum Beispiel Taschenrechnern, vergleichbar. Ähnlich der Methoden-Schnittstellen von Klassen gibt es bei Taschenrechnern verschiedene Bedienungs-Schnittstellen. Am gebräuchlichsten sind die algebraische Eingabelogik, die umgekehrte polnische Notation und die Eingabe mit BASIC-Befehlen. Abbildung 1-4 zeigt die Vererbungshierarchie dieser Schnittstellen.

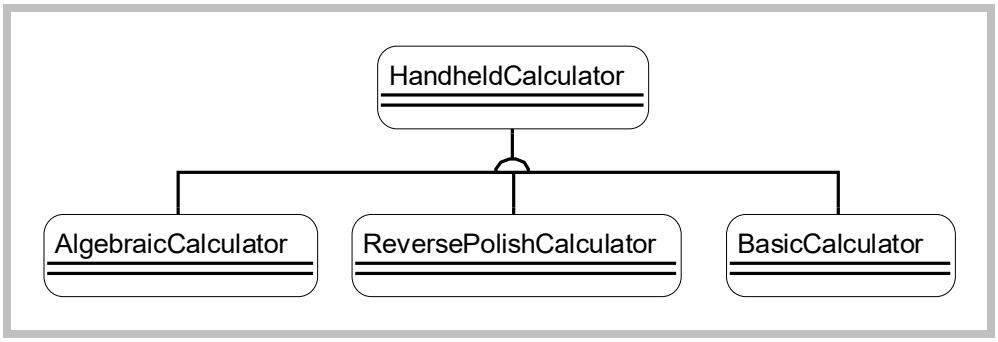

Abb. 1-4: *Taschenrechnergruppen nach Eingabelogik*

Löst man mit solchen Taschenrechnern viele Aufgaben und wechselt dabei oft die Eingabelogik, kommt es zu häufigen Bedienungsfehlern. Weniger kritisch ist hingegen die Verwendung mehrerer Taschenrechner, die mit derselben Eingabelogik arbeiten und sich nur im Funktionsumfang unterscheiden (siehe Abbildung 1-5).

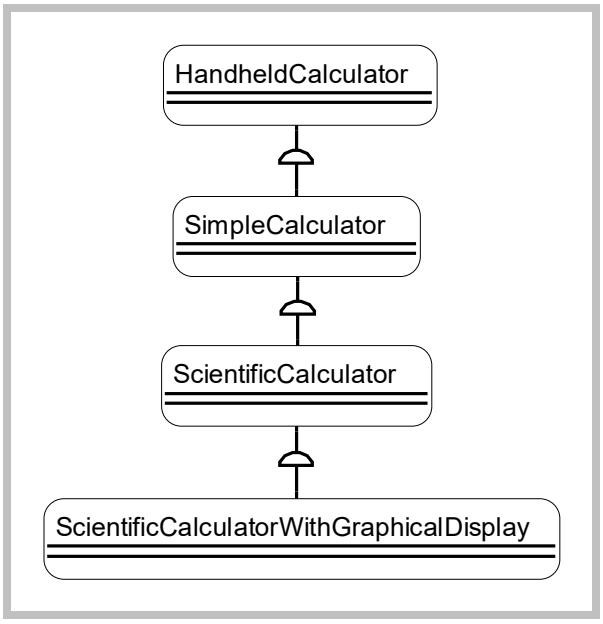

Abb. 1-5: *Taschenrechner mit wachsendem Funktionsumfang*

Wechselt man häufig zwischen Taschenrechnern dieser Gruppen, wird man kaum Bedienungsfehler verursachen. Es kann lediglich dazu kommen, daß man auf einem Rechner eine Funktion vermißt, die nur auf einem anderen Gerät verfügbar ist. Ein Bedienungsfehler bei einem Taschenrechner ist meist offensichtlich und wird schnell bemerkt. Weniger angenehm ist ein Fehler beim Gebrauch einer Klasse. Dies ist ein Programmierfehler, der zum fehlerhaften Verhalten oder zum Abbruch des Anwendungsprogramms führen kann. Die Folge ist eine lange Fehlersuche. Damit uns solche Fehler nicht unterlaufen, richten wir uns im weiteren nach folgender Regel.

Ein übersichtliches Design eines Klassenbaums erhalten wir, indem wir die Anzahl der Interfaces aller Klassen minimieren, nicht die Anzahl der Klassen.

Diese Regel besagt, daß die Komplexität eines Klassenbaumes nicht so sehr von der Anzahl der Klassen abhängt, sondern von der Anzahl verschiedener Interfaces, die diese Klassen besitzen. Die beiden Vererbungsbäume für Taschenrechner enthalten jeweils vier Typen. Dennoch ist der zweite Vererbungsbaum in der Handhabung einfacher, weil die Interfaces nicht grundverschieden sind, sondern aufeinander aufbauen. Ein Klassenbaum mit 50 Klassen kann in der Handhabung einfacher als ein Baum mit 20 Klassen sein, wenn in ersterem nur insgesamt fünf verschiedene Interfaces vorkommen, in letzterem aber jede Klasse ein anderes Interface besitzt. Beim Hinzufügen neuer Klassen zu einem bestehenden Baum sollten möglichst vorhandene Interfaces genutzt werden, zum Beispiel durch Vererbung. Dann erhöhen die neuen Klassen die Funktionalität des Klassenbaumes, also seine Brauchbarkeit. Die Komplexität erhöht sich hingegen nur unwesentlich, da keine neuen Interfaces hinzukommen. Damit bleibt der Klassenbaum in der Handhabung einfach.

1.2.3 Brandschutz statt Feuerwehr

Ein Bauarbeiter erzählt: Auf dieser Baustelle hat es schon fünfmal gebrannt, aber ich arbeite gern hier. Das Fundament und der Rohbau waren schnell fertig. Es ist ja nicht das erste Haus, das wir bauen. Am Anfang gefiel mir die Arbeit nicht so gut. Es war alles noch so sauber, als sollte es ein Krankenhaus werden. Das änderte sich aber schnell, nachdem es draußen kalt wurde. Die Heizungsmonteure hatten das oberste Stockwerk übersehen. Wegen der Kälte nutzte ein Maler zum Trocknen der Farbe einen Fön und ging in die Baracke. Eine halbe Stunde später mußten wir zum ersten Mal die Feuerwehr rufen. Es war aber nicht so schlimm, wie es auf den ersten Blick aussah. Der zweite Brand brach einige Tage später im Erdgeschoß aus und war etwas schwieriger. Das Mauerwerk wurde in einem Zimmer stark beschädigt. Unser Chef, der als Student ein guter Baustatiker war, änderte daraufhin die Baupläne und ließ eine zusätzliche Wand einziehen...

Wir wünschen unserem Freund, daß er nicht eines Tages eine Verletzung erleidet. Seinem Chef wünschen wir, daß er für den fertigen Bau einen Hausherrn findet und nicht selbst darin wohnen muß. Beim Programmieren möchten wir jedoch ohne Feuerlöscher auskommen. Die Anwender sollen sich in unserem Software-Gebäude wohl fühlen. Deshalb betreiben wir auf unserer digitalen Baustelle rechtzeitig Brandschutz.

Informationsverarbeitende Prozesse im Menschen und Computer unterscheiden sich unter anderem dadurch, daß sie beim Menschen auch unter extremen Bedingungen selten außer Kontrolle geraten. Geben wir zum Beispiel dem Schaffner in der Eisenbahn aus lauter Verwirrung statt der Fahrkarte eine Diskette, wirft es diesen nicht gleich aus den Schuhen. Er prüft das ihm übergebene Objekt, stellt fest, daß es sich um einen verkehrten Objekttyp handelt, und gibt es mit einer Fehlermeldung an uns zurück. Anders ist das Übergeben eines falschen Objekts an eine Methode. Selten werden die Parameter geprüft. Dementsprechend häufig ist das unkontrollierte Verhalten eines Programms bei einem fehlerhaften Methodenaufruf. Solche Fehler kann man nach dem Feuerwehr-Prinzip beseitigen. Man startet einen Debugger und tastet sich schrittweise durch das Programm. Hat man die fehlerhafte Stelle entdeckt, berichtigt man das Programm. Kann man aber nach dieser Korrektur mit ruhigem Gewissen weiterarbeiten? Kann es nicht sein, daß eine Woche später dieselbe Methode wieder mit verkehrten Parametern aufgerufen wird? Wäre es nicht besser, man würde in die Methode zusätzliche Prüfungen einbauen?

Innerhalb eines Programms gibt es zahlreiche Stellen, an denen der weitere Verlauf von der Korrektheit bisheriger Ergebnisse abhängt. Manchmal ist die Richtigkeit der Zwischenergebnisse durch den Kontrollfluß gewährleistet. In solchen Fällen benötigen wir keine zusätzlichen Prüfungen. Oft liegt jedoch ein Fehlverhalten im Bereich des Möglichen oder Wahrscheinlichen. Dann sollten wir die Korrektheit der Daten sicherstellen, bevor wir mit ihnen weiterarbeiten. Diese Tests erfolgen meist mit dem Makro `ASSERT`. Es erhält als Parameter die zu prüfende Bedingung. Ist die Bedingung nicht erfüllt, wird eine detaillierte Fehlermeldung ausgegeben. Typische Stellen für das Prüfen zusätzlicher Bedingungen sind die Ein- und Austrittspunkte von Methoden. Am Anfang werden die übergebenen Parameter getestet. Am Ende muß sichergestellt werden, daß die Resultate die gewünschten Eigenschaften besitzen. Neben diesen Vor- und Nachbedingungen müssen auch Zwischenergebnisse geprüft werden. Sehen wir uns als Beispiel eine Methode an, die in einer Zeichenkette ein bestimmtes Zeichen durch ein anderes ersetzen soll.

```
void Replace (char * pc_string, char c_old, char c_new)
  {
  ASSERT (pc_string != 0); // Vorbedingungen
  ASSERT (c_old != '\0');
  ASSERT (c_new != '\0');
  char * pc_found = strchr (pc_string, c_old);
  while (pc_found != 0)
    {
    ASSERT (pc_found >= pc_string); // Zwischenbedingungen
```

```
    ASSERT (pc_found < pc_string + strlen (pc_string));
    * pc_found = c_new;
    pc_found = strchr (pc_string, c_old);
    }
  ASSERT (strchr (pc_string, c_old) == 0); // Nachbedingung
  }
```

In dieser kleinen Methode wurde das ASSERT-Makro etwas übertrieben eingesetzt. Normalerweise sind Zusatzbedingungen nur nötig, wenn ein begründeter Verdacht auf ein Fehlverhalten vorliegt. ASSERT-Makros können auch bei mäßiger Verwendung einen Geschwindigkeitsverlust verursachen. Deshalb sollten sie nur in der Testphase des Programms wirksam sein. Bevor wir das getestete Programm dem Anwender übergeben, definieren wir beim Übersetzen das Makro NDEBUG. Dann expandieren alle ASSERT-Makros zu einer leeren Zeichenkette und belasten nicht mehr die Rechenzeit. Die Definition des ASSERT-Makros kann zum Beispiel so aussehen:

```
#ifdef NDEBUG
  #define ASSERT(condition)
#else
  #define ASSERT(condition) \
    if (! (condition)) InternalError (#condition, __FILE__, __LINE__)
#endif
```

Das ASSERT-Makro führt nicht nur zu einem robusteren Programm, es hat auch weitere Vorteile. Formulieren wir zusätzliche Bedingungen, durchdenken wir den Kontext besser, auf den sie sich beziehen. Bei Vor- und Nachbedingungen ist es der Kontext, in dem die Methode verwendet werden kann. Bei Zwischenbedingungen ist es der Kontext, in dem andere Methoden aufgerufen werden. Dieses tiefere Durchdenken macht uns auf Fehlersituationen aufmerksam, noch bevor das Programm zum ersten Mal gestartet wurde. Für den Anwender der Methode sind besonders die Vorbedingungen wichtig. Sie zeigen ihm knapp und korrekt, wie die Methode aufzurufen ist. Zusatzbedingungen im Quelltext sind eine Art Dokumentation und erleichtern auch die Pflege des Programms. Die folgende Regel faßt diese Erkenntnisse zusammen.

Durch Zusatzbedingungen erhalten wir ein robusteres und besser durchdachtes Programm. Gleichzeitig wird der Programmtext dokumentiert.

1.2.4 Vorsicht, Falle im Unsichtbaren

Die Programmiersprache C++ weist gegenüber C zahlreiche Erweiterungen auf. Die wichtigsten Neuerungen unterstützen objektorientierte Konzepte. Dazu zählen Vererbung, virtuelle Methoden und die Vergabe von Zugriffsrechten für die Elemente einer Klasse. Andere Erweiterungen sind programmtechnischer Natur, zum Beispiel Inline-Methoden, das Überladen von Methoden und das Definieren eigener Operatoren. Die neue Funktionalität erhöht in vielen Fällen den Programmierkomfort. Einige Eigenschaften der Sprache C++ müssen wir jedoch beim Programmieren besonders beachten.

Ein C-Programm kann fast linear in die Maschinensprache des Computers übersetzt werden. Der Compiler fügt beim Übersetzen nur wenige Anweisungen hinzu. Wir sehen nahezu alles, was zur Laufzeit im Computer abläuft, auch in unserem Programmtext. Diese Transparenz ist besonders wichtig, wenn im Programm ein Fehler gesucht werden muß, und erleichtert das Optimieren. In einem C++-Programm laufen hingegen viele Prozesse im Unsichtbaren ab. Der Compiler liest zwischen den Programmzeilen Dinge heraus, die ihm der Sprachstandard vorschreibt. Ein Programmierer, der mit der Sprache C++ noch wenig vertraut ist, kann dabei Wichtiges übersehen. Die unsichtbaren Anweisungen, die der

Compiler hinzufügt, beeinflussen manchmal die Performance. Damit werden wir uns später ausführlich beschäftigen. Manchmal ändern diese Zusätze das Verhalten des Programms. Diese Fälle müssen wir jetzt schon behandeln, um das nachfolgende Beispielprogramm fehlerfrei zu implementieren.

Jede C++-Klasse enthält mindestens einen Konstruktor, mindestens einen Gleich-Operator und genau einen Destruktor. Definieren wir diese Methoden nicht selbst, werden sie vom Compiler erzeugt. Die Regeln, nach denen sie automatisch generiert werden, sind nicht gerade einfach und veranlassen auch den Profi, immer wieder in der Sprachdefinition nachzuschauen. Ein Standard-Konstruktor ist ein Konstruktor, der keine Parameter oder nur Parameter mit Vorgabewerten besitzt. Ein Kopier-Konstruktor ist ein Konstruktor, der mit einem einzelnen Objekt derselben Klasse aufgerufen werden kann. Ist überhaupt kein Konstruktor definiert, generiert der Compiler einen Standard-Konstruktor ohne Parameter. Der Kopier-Konstruktor wird automatisch erzeugt, wenn kein anderer Kopier-Konstruktor definiert ist. Analog verhält sich der Gleich-Operator. Einen Destruktor gibt es nur einmal. Definieren wir ihn nicht selbst, wird er generiert.

Der Compiler erzeugt nicht nur Methoden-*Definitionen* sondern auch implizite Methoden-*Aufrufe*. Zum Beispiel werden in einem Konstruktor die Konstruktoren der Basisklassen und der Attribute automatisch aufgerufen. Jedes temporäre Objekt wird implizit mit einem Konstruktor erzeugt und mit dem Destruktor zerstört. Sehen wir uns ein Beispiel an, aus dem der Compiler wesentlich mehr macht, als auf dem Papier steht. Es ist ein Prinzipbeispiel, nicht eine besonders elegante Berechnung der Fakultät.

```
class ct_Number
  {
  long       l_Value;
public:
  ct_Number  Factorial ();
  ....
  };

ct_Number ct_Number:: Factorial ()
  {
  ct_Number co_result = * this;
  if (l_Value > 1)
    {
    co_result. l_Value --;
    co_result = co_result. Factorial ();
    co_result. l_Value *= l_Value;
    }
  return co_result;
  }
```

In der Klasse `ct_Number` erzeugt der Compiler automatisch den Standard-Konstruktor `ct_Number ()`, den Kopier-Konstruktor `ct_Number (const ct_Number &)`, den Destruktor `~ct_Number ()` und den Gleich-Operator `ct_Number & operator = (const ct_Number &)`. Daß wir diese Methoden nicht selbst definiert haben, ist bei einer so einfachen Klasse nicht relevant. Interessant ist jedoch, daß der Compiler die generierten Methoden auch implizit aufruft. Die Methode `Factorial` liefert ein Objekt der Klasse `ct_Number`. Da keine Adresse (Zeiger oder Referenz) zurückgegeben wird, erzeugt der Compiler in der `return`-Anweisung durch den Kopier-Konstruktor ein temporäres Objekt. In der Definition der lokalen Variablen `ct_Number co_result = * this` wird ebenfalls der Kopier-Konstruktor aufgerufen. Die Schreibweise mit Gleichheitszeichen ist irreführend und identisch mit `ct_Number co_result (* this)`. In der Anweisung `co_result = co_result. Factorial ()` wird das temporäre Objekt, das ein anderer Aufruf der Methode `Factorial` liefert, mit dem Gleich-Operator der lokalen Variablen zugewiesen und anschließend mit dem Destruktor zerstört.

An diesem Beispiel sehen wir, daß die genannten Methoden in einem Programm häufig verwendet werden können, auch wenn sie an keiner Stelle explizit aufgerufen werden.

Haben wir vergessen, diese Methoden für eine Klasse zu definieren, kann es bei der Anwendung dieser Klasse zu Fehlern kommen. Deshalb müssen wir jede Klasse daraufhin prüfen, ob die generierten Methoden den gewünschten Effekt haben. Zum Beispiel ruft der generierte Kopier-Konstruktor die Kopier-Konstruktoren der Basisklassen und der Attribute auf. Primitive Datentypen (`int`, `char` usw., auch Zeiger) werden binär kopiert. Entspricht dieses Verhalten nicht unseren Erwartungen, müssen wir den Kopier-Konstruktor selbst definieren. Die folgende Regel verdeutlicht noch einmal das so eben Behandelte.

Wir prüfen bei der Implementierung jeder Klasse, ob die automatisch generierten Methoden Standard-Konstruktor, Kopier-Konstruktor, Destruktor und Gleich-Operator den gewünschten Effekt haben. Ist das nicht der Fall, müssen wir diese Methoden selbst definieren.

1.3 Ein Beispielprogramm

1.3.1 Beschränkung auf das Wesentliche

Moderne, interaktive Programme bestehen aus zahlreichen Komponenten. Mit der Benutzeroberfläche können wir Informationen ansehen und Aktionen auslösen. Sie greift dabei auf interne Daten zu. Die prinzipielle Architektur interaktiver Programme wird am besten durch das **Model-View-Controller-Konzept** beschrieben (siehe Abbildung 1-6).

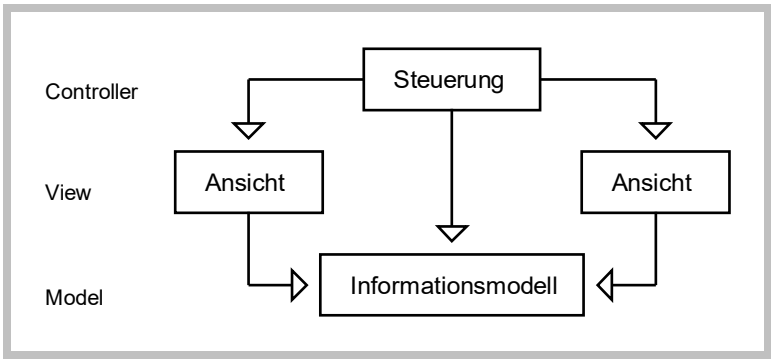

Abb. 1-6: *MVC-Architektur eines interaktiven Programms*

Das MVC-Konzept entstand in der Smalltalk-Gemeinschaft. Es ist heute allgemein anerkannt und wird von vielen objektorientierten Software-Entwicklern eingesetzt. Die Grundidee besteht in der Trennung des Informationsmodells von Sichten darauf und von der Benutzerschnittstelle. Das Informationsmodell (*Model*) enthält das fachliche Wissen und die eigentlichen Daten des Programms. Ist es unabhängig von seinen Darstellungen, kann es in beliebige Softwareumgebungen integriert werden. Zum Beispiel ist es dann möglich, dieselben Daten interaktiv oder in einem Batchlauf zu bearbeiten. Auf die internen Daten können verschiedene Ansichten (*Views*) gebildet werden. Diese sollten möglichst unabhängig voneinander sein, damit die Änderung oder Hinzunahme einer Sicht die anderen nicht beeinflußt. Der interaktive Teil eines Programms wird in der Benutzerschnittstelle (*Controller*) zusammengefaßt. Ist er unabhängig von den anderen Teilen, kann er leicht an neue Erfordernisse oder ein anderes Betriebssystem angepaßt werden.

Die Performance datenintensiver Programme wird im wesentlichen durch das zugrundeliegende **Informationsmodell** bestimmt. Dort werden die meisten Ressourcen verbraucht (Speicherplatz und Rechenzeit). Deshalb werden wir uns im folgenden auf die internen Daten konzentrieren. Die Betrachtung des Informationsmodells ist allgemeingültig

und auf alle C++-Programme anwendbar. Wir benötigen dazu lediglich Kenntnisse der Programmiersprache und ein wenig Wissen über den Compiler. Die Anbindungen an Datenbanken, Betriebssystem und Benutzeroberfläche bleiben außer Betracht. Diese Komponenten sind aus dem Blickwinkel des reinen C++-Programms externe Schnittstellen. Ihre Performance können wir durch intensives Studium der zugehörigen Handbücher beeinflussen, aber kaum durch die Programmiersprache.

1.3.2 Aufgabenstellung des Programms

Programmtechnische Konzepte, wie man sie zum Beispiel für die Erhöhung der Performance einsetzt, sind meist sehr abstrakt. Um diese Abstraktionen besser verstehen zu können, sehen wir sie uns anhand eines praktischen Beispiels an. Bei der Auswahl eines geeigneten Programmbeispiels stehen wir vor einem schwierigen Problem. Die Verbesserung der Performance ist oft nur bei großen Programmen mit komplex strukturierten Daten sinnvoll. Solche Dinosaurier-Programme sind ein willkommenes Fressen für einen Programm-Optimierer. Sie können aber im Rahmen eines Buches nicht ausführlich behandelt werden, denn die meisten Leser haben keinen Appetit auf Dinosaurier. Wir sehen uns also nach einem kleineren Beispiel um. Dabei reicht uns ein Prinzip-Beispiel, denn die Prinzipien zur Optimierung eines C++-Programms sind der eigentliche Gegenstand dieses Buches.

Fast jeder kennt die Hilfesysteme moderner Programme und Betriebssysteme mit graphischer Benutzeroberfläche. In einem solchen Hilfesystem ist eine Reihe von Themen untergebracht. Zu jedem Thema gibt es einen Text. Dieser ist kein normaler, fließender Text, wie wir ihn von gedruckten Medien kennen, sondern ein **Hypertext**, ein Text mit Hyperlinks. Diese Hyperlinks erleichtern uns wesentlich den Umgang mit dem Hilfesystem. Durch sie gelangen wir mit wenigen Tasten zu verwandten Themen, die das ursprüngliche Thema weiter vertiefen. Außerdem sind einige Stellen im Text hervorgehoben, zum Beispiel durch Fettschrift, Kursivschrift oder Unterstreichung. In Abbildung 1-7 sehen wir ein Beispiel mit drei Themen, wobei das eine Thema Verweise auf die anderen Themen enthält. Die umrandeten Worte stellen Hyperlinks dar.

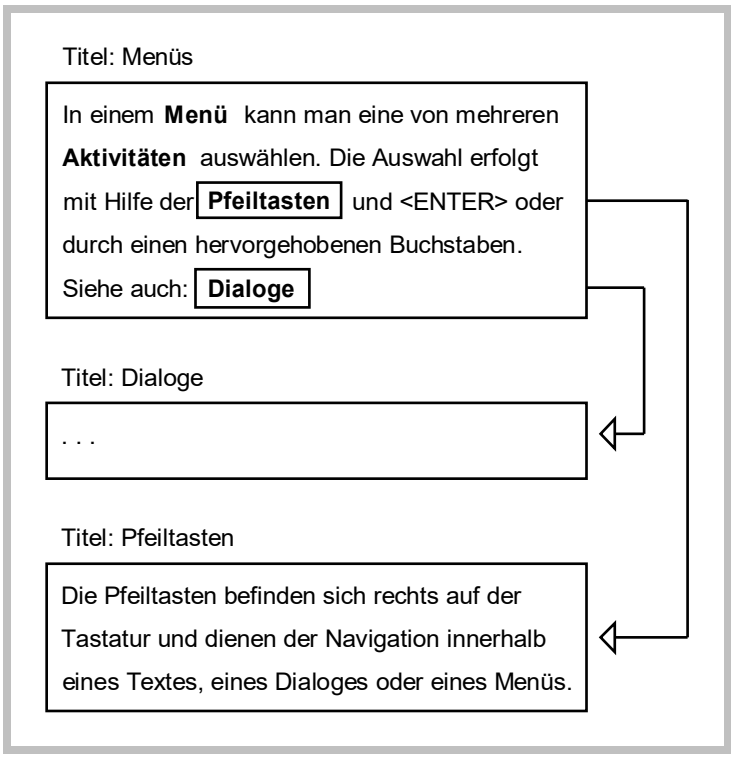

Abb. 1-7: *Beispiel für einen Hypertext*

Moderne Hilfesysteme verfügen über vielfältige Möglichkeiten zur Darstellung von Informationen und zum Navigieren. Dementsprechend komplex sind die darunter liegenden Datenstrukturen. In unserem Beispiel beschränken wir uns auf das Wesentliche. Unser Hilfesystem soll nur einfachste Funktionen beherrschen, und wir betrachten nur sein Informationsmodell. Das Programmbeispiel, das wir im folgenden behandeln werden, ist also der *objektorientierte Kern eines einfachen Hilfesystems*, deshalb nennen wir es **OHelp**. Die wichtigste Aufgabe eines Programmkerns ist die Verwaltung des Informationsmodells. OHelp soll einen Hypertext verwalten. Daran stellen wir die folgenden Anforderungen:

- Ein Hypertext hat einen Namen, besteht aus mehreren Themen und hat einen Verweis auf ein Wurzelthema, das den Einstieg in das Hilfesystem ermöglicht.
- Zu einem Thema gehören ein Name und ein Text.
- Beliebige Textstellen können mit einer Formatierung versehen werden (siehe "Menü" und "Aktivitäten" im Beispiel).
- Ein Thema kann Verweise auf andere Themen haben (Hyperlinks). Ein Hyperlink hat optional eine Textposition.
- Ein Hyperlink mit Textposition dient der Anzeige im fließenden Text (siehe "Pfeiltasten" im Beispiel).
- Hyperlinks ohne Textposition können am Ende des Textes aufgelistet werden (siehe "Dialoge" im Beispiel).

1.4 Fundamentale Klassen von OHelp

Ein Programm besteht aus mehreren Schichten, die aufeinander aufbauen. Die unterste Schicht jedes C++-Programms ist die Programmiersprache. Diese ist vorgegeben und kann von uns nicht beeinflußt werden. Die nächste Schicht ist die C-Standardbibliothek, die mit geringen Änderungen von den C++-Compilern übernommen wurde. Auch diese ist als langjähriger Standard vorgegeben. Es lohnt sich nicht, in dieser Programmierebene Änderungen vorzunehmen, denn die C-Standardbibliothek ist auf vielen Plattformen verfügbar und macht das Programm weitestgehend portabel. Allerdings sind die Module dieser Bibliothek in C geschrieben und keine befriedigende Grundlage für die Erstellung objektorientierter Programme. Es fehlen einige grundlegende Module, die wir für die Entwicklung jedes Programms benötigen, zum Beispiel die Mengen (*Collections*). Es ist also sinnvoll, auf das C-Laufzeitsystem eine eigene Schicht zu bauen. In dieser Schicht sind allgemeingültige **fundamentale Klassen** enthalten, die von mehreren Programmen genutzt werden können.

Wie bei der Auswahl eines geeigneten Beispielprogramms beschränken wir uns auch beim Entwurf der fundamentalen Klassen auf das Wesentliche. Moderne, objektorientierte Klassenbibliotheken enthalten meist 50, 100 oder noch mehr Klassen und neigen zur Unübersichtlichkeit. Im Rahmen dieses Buches soll keine vollständige Klassenbibliothek behandelt werden. Wir begnügen uns mit den fundamentalen Klassen, die in unserem Beispielprogramm benötigt werden. Beim Durchsehen der oben aufgelisteten Anforderungen an OHelp stellen wir fest, daß an einigen Stellen Mengen und Zeichenketten (*Strings*) benötigt werden. So enthält ein Hypertext eine Menge Themen, ein Thema enthält eine Menge Formatierungen und Hyperlinks. Hypertext und Thema haben einen Namen, und das Thema enthält den Text.

1.4.1 Design der fundamentalen Klassen

In einem Anwendungsprogramm gibt es viele verschiedene Anforderungen an Collections. Diese Anforderungen lassen sich grob in zwei Bereiche gliedern, die Schnittstelle (*Interface*) und die Implementierung. In diesem Abschnitt wollen wir uns mit dem Interface der fundamentalen Klassen, also auch der Collections, beschäftigen. Dieses Interface stellt eine bestimmte Funktionalität zur Verfügung.

In bezug auf die Reihenfolge unterscheiden wir geordnete und ungeordnete Collections. In einer geordneten Collection haben die Elemente eine bestimmte Reihenfolge, während in einer ungeordneten Collection die Reihenfolge der Elemente zufällig sein kann. Innerhalb der Gruppe der ungeordneten Collections gibt es Sets und Bags. In einem Set kann ein Element nur einmal vorkommen, in einem Bag kann hingegen dasselbe Element mehrmals enthalten sein. Diese Collections entsprechen dem Klassenbaum in Abbildung 1-8.

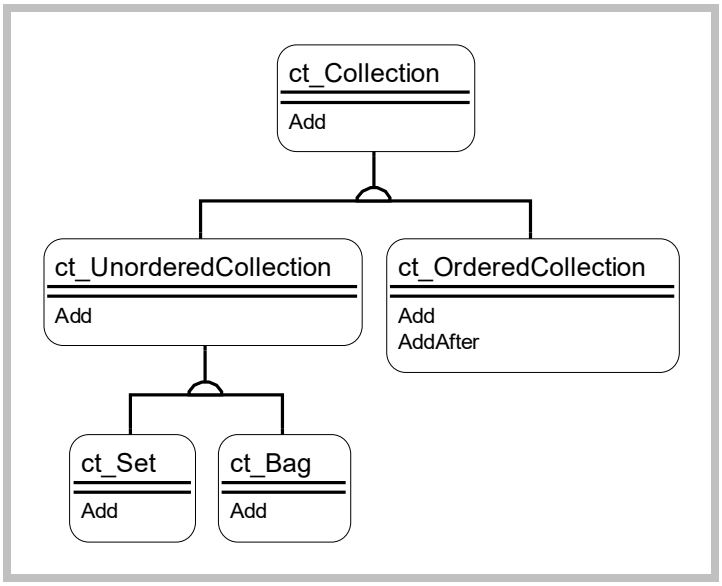

Abb. 1-8: *Verbreitete Collections-Hierarchie*

Im Abschnitt 1.2.2 "Überblick ist Alles" haben wir uns überlegt, wie wir die Übersichtlichkeit eines Klassenbaums erhöhen. Sehen wir uns unter diesem Blickwinkel die obige Collections-Hierarchie einmal näher an. Die Klassen `ct_Set` und `ct_Bag` unterscheiden sich nur durch die Methode für das Einfügen eines neuen Elements. Bei der Klasse `ct_Set` wird das Element nur hinzugefügt, wenn es noch nicht in der Collection enthalten ist. Bei der Klasse `ct_Bag` unterbleibt diese Prüfung. Lohnt es sich, wegen dieses geringen Unterschieds zwei verschiedene Klassen mit einem semantisch verschiedenen Interface zu deklarieren? Oder ist es nicht einfacher, dafür zwei Methoden in dieselbe Klasse aufzunehmen, zum Beispiel `Add` (Hinzufügen) und `AddCond` (Bedingtes Hinzufügen)? Im letzteren Fall haben wir gleich zwei Interfaces eingespart.

Betrachten wir nun die geordneten und die ungeordneten Collections. Der Unterschied besteht im Grunde nur darin, daß eine geordnete Collection etwas mehr Funktionalität aufweist. In einer ungeordneten Collection gibt es zum Einfügen eines neuen Elements nur die Methode `Add` ohne Positionsangabe. Bei einer geordneten Collection können wir die Stelle genau angeben, an der das neue Element hinzugefügt werden soll. Weiter können wir voraussetzen, daß bei jedem Durchlaufen der geordneten Collection die Reihenfolge der Elemente dieselbe ist. Auch in diesem Falle lohnt es sich nicht, zwei verschiedene Interfaces zu deklarieren. Einfacher ist es, nur eine Klasse zu verwenden, und in diese Klasse die Methoden `Add` und `AddAfter` aufzunehmen. Die abstrakte Basisklasse für die Collections in unserer kleinen Bibliothek hat also das Interface einer geordneten Collection mit

Erweiterungen. Alle konkreten Implementierungen von Collections werden dasselbe Interface haben.

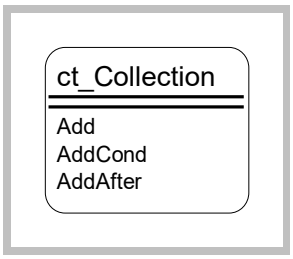

Abb. 1-9: *Abstrakte Basisklasse für Collections*

Die Klasse in Abbildung 1-9 enthält die gesamte Funktionalität der weiter oben dargestellten Vererbungshierarchie für Collections. Sie umfaßt sowohl das Interface einer ungeordneten als auch das einer geordneten Collection (`Add` und `AddAfter`). Sie kann neue Elemente wie eine Bagcollection (`Add`) oder eine Setcollection (`AddCond`) aufnehmen. Wir haben im Klassenbaum fünf Interfaces auf eines reduziert und damit dessen Handhabung vereinfacht.

Eine polymorphe Collection enthält normalerweise Zeiger (*Pointer*) auf ihre Elemente. In der strukturierten Programmierung war es üblich, dafür untypisierte Zeiger zu verwenden. Es lag in der Verantwortung des Programmierers, diese richtig einzusetzen. In C++ gebraucht man dazu meist eine abstrakte Basisklasse, nennen wir sie `ct_Object`. Die Collection speichert Zeiger auf diese Basisklasse. Alle Klassen, deren Objekte in eine Collection gelangen sollen, müssen von der abstrakten Basisklasse erben. Jedesmal, wenn auf ein Element einer Collection zugegriffen werden soll, muß ein Zeiger auf die Basisklasse `ct_Object` in einen Zeiger auf die abgeleitete Klasse umgewandelt werden. Deshalb muß die Basisklasse einen geeigneten Downcast-Mechanismus bereitstellen (*Cast* = Typumwandlung, *Downcast* = Cast zu einer abgeleiteten Klasse). Im neuesten C++-Standard sind dafür die Laufzeit-Typinformationen vorgesehen (***R**untime **T**ype **I**nformations*, kurz RTTI). Da die RTTI aber noch nicht von jedem Compiler unterstützt werden, benutzen wir einen eigenen Mechanismus.

Wir definieren in der Klasse `ct_Object` eine virtuelle Methode `GetTypeName`. Diese Methode muß in jeder abgeleiteten Klasse redefiniert werden und liefert einen Zeiger auf eine Zeichenkette, die den Namen dieser Klasse enthält. Weiterhin ist in der Klasse `ct_Object` eine nicht-virtuelle Methode `IsOfType` enthalten. Diese Methode erwartet als Parameter einen Zeiger auf einen Klassennamen und vergleicht diesen mit dem Resultat von `GetTypeName`. Sind beide Zeichenketten gleich, so ist das Objekt vom angegebenen Typ, und `IsOfType` liefert den Wahrheitswert `true`. Danach kann ohne Bedenken der Downcast vorgenommen werden.

Im neuesten C++-Standard existieren für Wahrheitswerte der Datentyp `bool` und die Konstanten `false` und `true`. Auch diese Erweiterungen werden noch nicht von jedem Compiler unterstützt. Wir nehmen sie explizit in unser Programm auf. Betrachten wir nun die Deklaration und eine Anwendung der Klasse `ct_Object`.

```
typedef int bool;
const bool false = 0;
const bool true  = 1;

class ct_Object
  {
public:
  virtual              ~ct_Object () { }
  virtual const char * GetTypeName () const = 0;
  bool                 IsOfType (const char * pc_typeName) const;
  };
```

```
bool ct_Object:: IsOfType (const char * pc_typeName) const
  {
  return strcmp (pc_typeName, GetTypeName ()) == 0;
  }

const char * ObjectToString (ct_Object * pco_obj)
  {
  ASSERT (pco_obj-> IsOfType ("ct_String"));
  ct_String * pco_string = (ct_String *) pco_obj;
  return pco_string-> GetStr ();
  }
```

Es gibt sicher noch leistungsfähigere Downcast-Mechanismen; für unser Beispielprogramm OHelp ist der geschilderte aber ausreichend. Wir müssen nur darauf achten, daß die Methode `IsOfType` einen Zeichenketten-Vergleich enthält und somit den Downcast verlangsamt. Sie ist eine Brandschutzmaßnahme und sollte nur innerhalb von `ASSERT`-Makros aufgerufen werden.

Für Einfügen, Löschen und Iterieren der Elemente einer Collection existieren verschiedene Techniken, die zum Teil erhebliche Laufzeitunterschiede aufweisen. Nutzen wir zum Beispiel zum Entfernen eines Objekts aus einer Collection einen Zeiger auf dieses Objekt, muß der Zeiger erst in der Collection gesucht werden. Beim einmaligen Entfernen mag dies unkritisch sein. Wird mit der Collection aber häufig gearbeitet, kann dadurch eine merkliche Verlangsamung eintreten. Die Collections, die wir gerade entwerfen, werden später einem kritischen Performancetest unterworfen. Es ist leicht, Mängel in solchen Klassen zu finden, die schon im Design unzureichend sind. Deshalb achten wir von vornherein auf eine gute Performance und statten unsere Collections mit einer effizienten Technik aus.

Zum Beschleunigen des Zugriffs auf die Objekte erhält jeder neue Eintrag eine eindeutige Identität, eine **EntryId**. Diese ist mit dem Index in einem Array vergleichbar. Den zugehörigen Datentyp nennen wir `t_EntryId`. An ihn stellen wir nur die eine Forderung, daß ein Element vom Typ `t_EntryId` mit dem numerischen Wert Null verglichen werden kann. Jede Implementierung einer Collection muß sicherstellen, daß der Wert Null nicht als eine gültige EntryId verwendet wird. Die Methode zum Einfügen eines Objekts liefert als Resultat einen neuen Wert vom Typ `t_EntryId`. Diese EntryId können wir zum schnelleren Zugriff auf das Objekt speichern. Einen Zeiger auf das eigentliche Objekt erhalten wir mit der Methode `GetObj`, die als Parameter eine EntryId erwartet. Zum Iterieren der Collection verwenden wir die Methoden `First` und `Next`. Auch diese Methoden liefern einen Wert des Typs `t_EntryId`. Am Ende der Collection erhalten wir den Wert Null. Mit der Methode `Delete` können wir einen Eintrag aus der Collection entfernen. Diese Methode erhält als Parameter eine EntryId, so daß das zu löschende Objekt nicht erst gesucht werden muß.

Ein häufiges Problem bei der Arbeit mit Collections ist das Ändern der Collection, während sie durchlaufen wird. In einigen Klassenbibliotheken wird dieses Problem ignoriert. In anderen gibt es aufwendige Iterator-Objekte, die zwar das gewünschte Verhalten zeigen, aber nur mit einem unangemessen hohen Aufwand, der sich in der Performance niederschlägt. Durch eine kleine Änderung in dem oben beschriebenen Konzept lösen wir dieses Problem auf sehr einfache Weise. Wir fordern nur, daß die Methode `Delete` eine EntryId zurückgibt, und zwar die EntryId des nächsten Objekts. Nun können wir eine Collection iterieren und dabei beliebige Änderungen vornehmen. Das folgende Programmfragment verdeutlicht diese Vorgehensweise.

```
ct_Collection * pco_coll = ....;
// Abfrage der EntryId vom ersten Element:
t_EntryId o_currId = pco_coll-> First ();
// Der Wert Null bedeutet das Ende der Collection:
while (o_currId != 0)
  {
  if (pco_coll-> GetObj (o_currId)-> ....)
    // Übergang zum nächsten Element:
```

```
    o_currId = pco_coll-> Next (o_currId);
  else
    // Entfernen des aktuellen und Übergang zum nächsten Element:
    o_currId = pco_coll-> Delete (o_currId);
  }
```

Auf der Wunschliste für eine leistungsfähige Collection stehen natürlich noch weitere Methoden. Zum Beispiel könnten wir mit `Last` und `Prev` die Collection rückwärts durchlaufen. Wir wollen aber in die fundamentalen Klassen nur die Funktionalität aufnehmen, die wir im Beispielprogramm OHelp benötigen. Deshalb statten wir auch unsere Stringklasse nur mit den wichtigsten Methoden aus. Dazu zählen die Abfrage der Zeichenkette und ihrer Länge (`GetStr` und `GetLen`), jeweils eine Methode zum Einfügen und Löschen (`Insert` und `Delete`) und eine Methode, mit der wir auf ein einzelnes Zeichen zugreifen können (`operator []`). Abbildung 1-10 zeigt die wesentlichen Resultate unseres Designs.

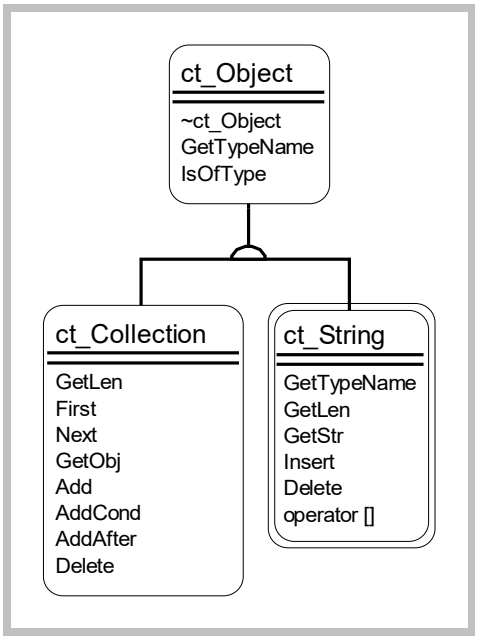

Abb. 1-10: *Design der fundamentalen Klassen*

1.4.2 Implementierung der Collections

Bei der Implementierung der fundamentalen Klassen beginnen wir wieder mit den Collections, denn diese beanspruchen wesentlich mehr Aufmerksamkeit als die Stringklasse. Im Design haben wir eine abstrakte Basisklasse für Collections entworfen. Nun müssen wir deren Methodennamen um Rückgabewert und Parameterliste ergänzen und sie in C++ aufschreiben. Zu dem schon bekannten Typ `t_EntryId` kommt ein weiterer hinzu, `t_Length`. Den Längentyp benötigen wir für die Anzahl der Elemente in einer Collection. Die beiden Typen `t_EntryId` und `t_Length` definieren wir erst, wenn die konkreten Collections implementiert sind, denn diese Typen müssen auf alle Collections passen.

```
class ct_Collection: public ct_Object
  {
public:
  virtual t_Length     GetLen () const = 0;
  virtual t_EntryId    First () const = 0;
  virtual t_EntryId    Next (t_EntryId o_id) const = 0;
  virtual ct_Object *  GetObj (t_EntryId o_id) const = 0;
  t_EntryId            Add (ct_Object * pco_obj);
  t_EntryId            AddCond (ct_Object * pco_obj);
```

```
  virtual t_EntryId    AddAfter (t_EntryId o_id, ct_Object * pco_obj) = 0;
  virtual t_EntryId    Delete (t_EntryId o_id) = 0;
  };
```

Die beiden Methoden `Add` und `AddCond` sind nicht virtuell. Wir können sie bereits in der Basisklasse `ct_Collection` definieren, denn sie sind von einer konkreten Implementierung unabhängig. Alle anderen Methoden sind rein virtuell und müssen in den abgeleiteten Klassen definiert werden. Für die Definition der Methode `Add` fordern wir, daß die Methode `AddAfter` die Position 0 akzeptiert.

```
t_EntryId ct_Collection:: Add (ct_Object * pco_obj)
  {
  return AddAfter (0, pco_obj);
  }

t_EntryId ct_Collection:: AddCond (ct_Object * pco_obj)
  {
  for (t_EntryId o_id = First (); o_id != 0; o_id = Next (o_id))
    if (GetObj (o_id) == pco_obj)
      return o_id;
  return Add (pco_obj);
  }
```

Die abstrakte Basisklasse `ct_Collection` gibt das Interface, die Methodenschnittstelle, vor. Nun gilt es, konkrete Collections zu schaffen. Es gibt zahlreiche Konzepte für die Implementierung von Collections. Dazu zählen Feld (*Array*), einfach und doppelt verkettete Liste (*Single*, *Double Linked List*), binärer Baum (*Binary Tree*) und Hash-Tabelle (*Hash Table*). Im Rahmen unserer kleinen Bibliothek fundamentaler Klassen werden wir uns wieder auf das Wesentliche beschränken und wählen diejenigen Collections aus, die in der Praxis am häufigsten verwendet werden. Das sind **Array** und **DList** (Double Linked List). Das Array-Konzept steht für optimale Speicherauslastung, das DList-Konzept für höheren Programmierkomfort.

Zunächst implementieren wir eine Arrayklasse mit dem Namen `ct_Array`. Statische Arrays sind bereits in der Programmiersprache C++ enthalten. Sie werden definiert, indem hinter dem Namen des Objekts die Größe angegeben wird, zum Beispiel `ct_Object aco_Array [20]`. Dieses Array besitzt eine feste Länge. Für die Konkretisierung unserer Collection-Basisklasse benötigen wird jedoch ein dynamisches Array, das heißt ein Array, dessen Länge variieren kann. Dafür definieren wir in der Klasse `ct_Array` einen Zeiger auf einen Speicherblock variabler Länge. In diesem Block werden Zeiger auf die Klasse `ct_Object` untergebracht. Das neue Attribut hat also die Definition `ct_Object * * ppco_Array`. Weiterhin benötigen wir ein Attribut `o_Length`, das die Anzahl der Einträge des Arrays enthält. Die Größe des dynamischen Speicherblocks erhalten wir durch `o_Length * sizeof (ct_Object *)`. In diesem Speicherblock sind die Zeiger auf die Objekte kompakt untergebracht. Unsere Arraycollection ist also speicherplatzoptimal.

Für Einfügen und Entfernen der Elemente eines dynamischen Arrays gibt es mehrere Konzepte. Wird ein Element entfernt, so kann die Stelle mit einem Nullzeiger belegt werden. Beim Iterieren des Arrays werden solche Stellen übergangen. Sollen neue Elemente hinzugefügt werden, so werden zuerst die Nullzeiger ersetzt, dann wird angefügt. Diese Vorgehensweise hat zwei wesentliche Nachteile. Zum einen kann die Reihenfolge der Elemente nicht beeinflußt werden, das heißt, es ist eine ungeordnete Collection. Zum anderen können im dynamischen Speicherblock Lücken entstehen, und der Vorteil der Speicherplatzoptimierung geht verloren. Deshalb entscheiden wir uns für das folgende Konzept. Beim Einfügen oder Löschen von Elementen kann die Position angegeben werden, und alle dahinter stehenden Einträge werden im Speicher verschoben. Das Speicherverschieben kostet zwar Zeit, aber wir verfügen nun über eine geordnete Collection mit einer optimalen Speicherauslastung. Ein Geschwindigkeitsverlust macht sich erst bei

sehr großen Arrays bemerkbar, denn moderne Computer besitzen schnelle Prozessorbefehle zum Speicherverschieben.

Die Größe eines Arrays ist durch den dynamischen Speicherblock begrenzt. Arbeiten wir mit einem 16-Bit-Compiler, kann ein Block maximal 64 KB umfassen. Darin können wir 16384 Zeiger unterbringen. Auch bei 32-Bit-Compilern existieren praktische Beschränkungen. Einige Betriebssysteme können nicht mehr als 1 MB zusammenhängenden Speicher bereitstellen. Die Klasse `ct_Array` ist eine konkrete Klasse. Wir vergessen also nicht, die folgenden wichtigen Methoden aufzunehmen: Konstruktor, Kopier-Konstruktor, Destruktor und Gleich-Operator. Zur Veranschaulichung der Methodenimplementierung ist im Programmausschnitt die Definition von `AddAfter` enthalten.

```
class ct_Array: public ct_Collection
  {
  t_Length             o_Length;
  ct_Object * *        ppco_Array;
public:
                       ct_Array ();
                       ct_Array (const ct_Array & co_init);
  virtual              ~ct_Array ();
  ct_Array &           operator = (const ct_Array & co_asgn);
  virtual const char * GetTypeName () const;
  virtual t_Length     GetLen () const;
  virtual t_EntryId    First () const;
  virtual t_EntryId    Next (t_EntryId o_id) const;
  virtual ct_Object *  GetObj (t_EntryId o_id) const;
  virtual t_EntryId    AddAfter (t_EntryId o_id, ct_Object * pco_obj);
  virtual t_EntryId    Delete (t_EntryId o_id);
  };

t_EntryId ct_Array:: AddAfter (t_EntryId o_id, ct_Object * pco_obj)
  {
  ASSERT (o_id <= o_Length);
  o_Length ++;
  ppco_Array = (ct_Object * *)
    realloc (ppco_Array, (unsigned) o_Length * sizeof (ct_Object *));
  ASSERT (ppco_Array != 0);
  if (o_id < o_Length - 1)
    memmove (ppco_Array + (unsigned) o_id + 1,
      ppco_Array + (unsigned) o_id,
      ((unsigned) (o_Length - o_id) - 1) * sizeof (ct_Object *));
  ppco_Array [(unsigned) o_id] = pco_obj;
  return o_id + 1;
  }
```

Widmen wir uns nun der doppelt verketteten Liste. Wir nennen die Collectionklasse `ct_DList`. Die Einträge einer DList sind durch Vorwärts- und Rückwärtsverweise miteinander verbunden. Wir können in der Liste sehr schnell Elemente einfügen und löschen. Die Größe der Liste ist nur durch den verfügbaren Hauptspeicher begrenzt. Die Klasse `ct_DListNode` dient der Speicherung eines einzelnen Eintrags. In einem Knoten (*Node*) befinden sich je ein Zeiger auf das Vorgänger- und Nachfolger-Node und natürlich ein Zeiger auf das Objekt, das Element der Liste ist. Da die Klasse `ct_DListNode` nur für den internen Gebrauch bestimmt ist, erbt sie nicht von `ct_Object`, besitzt nur private Member und deklariert die Klasse `ct_DList` als `friend`.

In der Klasse `ct_DList` benötigen wir ein Attribut `o_Length`, das die Anzahl der Einträge enthält, und ein Attribut `o_FirstNode` mit dem Verweis auf das erste Node. Auch die Klasse `ct_DList` ist eine konkrete Klasse. Zu den virtuellen Methoden, die von der Basisklasse `ct_Collection` ererbt werden, fügen wir Konstruktor, Kopier-Konstruktor, Destruktor und Gleich-Operator hinzu. Aus der Reihe der Methodenimplementierungen sehen wir uns als Beispiel wieder die Definition der Methode `AddAfter` an. In unserer Listenimplementierung

bilden die Nodes einen Ring. Auch das erste und letzte Element sind miteinander verbunden. Dadurch entfällt die Sonderbehandlung des Listenanfangs und -endes in `AddAfter` und `Delete`.

```
class ct_DListNode
  {
  friend class ct_DList;

  t_EntryId            o_PrevNode;
  t_EntryId            o_NextNode;
  ct_Object *          pco_Object;

                       ct_DListNode (ct_Object * pco_obj);
  };

class ct_DList: public ct_Collection
  {
  t_Length             o_Length;
  t_EntryId            o_FirstNode;

  ct_DListNode *       GetNode (t_EntryId o_id) const;
public:
                       ct_DList ();
                       ct_DList (const ct_DList & co_init);
  virtual              ~ct_DList ();
  ct_DList &           operator = (const ct_DList & co_asgn);
  virtual const char * GetTypeName () const;
  virtual t_Length     GetLen () const;
  virtual t_EntryId    First () const;
  virtual t_EntryId    Next (t_EntryId o_id) const;
  virtual ct_Object *  GetObj (t_EntryId o_id) const;
  virtual t_EntryId    AddAfter (t_EntryId o_id, ct_Object * pco_obj);
  virtual t_EntryId    Delete (t_EntryId o_id);
  };

t_EntryId ct_DList:: AddAfter (t_EntryId o_id, ct_Object * pco_obj)
  {
  t_EntryId o_new = (t_EntryId) new ct_DListNode (pco_obj);
  if (o_id == 0)
    {
    if (o_Length != 0)
      o_id = GetNode (o_FirstNode)-> o_PrevNode;
    o_FirstNode = o_new;
    }
  if (o_Length != 0)
    {
    t_EntryId o_next = GetNode (o_id)-> o_NextNode;
    GetNode (o_new)->  o_PrevNode = o_id;
    GetNode (o_new)->  o_NextNode = o_next;
    GetNode (o_id)->   o_NextNode = o_new;
    GetNode (o_next)-> o_PrevNode = o_new;
    }
  o_Length ++;
  return o_new;
  }
```

1.4.3 EntryId und Längenangabe

Nachdem wir die Klassen `ct_Array` und `ct_DList` deklariert haben, müssen wir noch die passenden Datentypen für Längenangabe und EntryId finden. Die Länge eines statischen Arrays in C++ ist bei den meisten Compilern durch den Datentyp `unsigned int` begrenzt. Dieselbe Begrenzung gilt auch für dynamische Arrays, die mit einem Zeiger verwaltet

werden, denn in C++ werden Arrays und Zeiger sehr ähnlich behandelt. Unsere Implementierung der DList hat hingegen keine Einschränkungen bezüglich der Anzahl der enthaltenen Elemente. Der dazu geeignete ganzzahlige Typ ist `unsigned long`. Das ist der umfassendere Datentyp, und ihn können wir für die Definition des Längentyps in der Basisklasse `ct_Collection` verwenden. Die EntryId eines neuen Eintrags im Array ist wieder vom Typ `unsigned int`, denn die EntryId ist identisch mit dem Index im Array. Beim Hinzufügen eines neuen Elementes in einer DList wird ein neues Node erzeugt. Der Zeiger auf dieses Node (`ct_DListNode *`) dient als EntryId. Für die allgemeine EntryId benötigen wir also einen Datentyp, der sowohl `unsigned int` als auch einen Zeiger enthalten kann. Die Größe dieser Datentypen ist abhängig vom Speichermodell, mit dem wir unser Programm übersetzen. Im allgemeinen gilt, daß ein Objekt vom Typ `unsigned long` sowohl einen `int`-Wert als auch einen Zeiger-Wert enthalten kann.

Würden wir als Datentyp für die Länge `unsigned int` verwenden, wäre die DList einer unnötigen Beschränkung ausgesetzt. Eine Instanz der Klasse `ct_DList` könnte nur so viele Elemente aufnehmen, wie der Wertebereich von `unsigned int` zuläßt. Durch die Verwendung von `unsigned long` als Längentyp müssen wir aber in der Implementierung der Methoden der Klasse `ct_Array` häufig casten, zum Beispiel in der folgenden Programmzeile.

```
ppco_Array = (ct_Object * *)
  realloc (ppco_Array, (unsigned) o_Length * sizeof (ct_Object *));
```

Diese häufigen Typumwandlungen könnten wir vermeiden, wenn wir statt `unsigned long` einen Klassentyp verwenden. Die Programmiersprache C++ bietet zahlreiche Möglichkeiten, primitive Datentypen (`int`, `char` usw.) durch Klassen zu ersetzen. Das Wichtigste ist, daß wir für eine Klasse eigene Operatoren definieren können. Eine Klasse für einen Längentyp könnte zum Beispiel so aussehen.

```
class ct_Length
  {
  union
    {
    unsigned       u_Length;
    unsigned long  ul_Length;
    };
public:
  ct_Length &      operator = (unsigned u_length)
                     { u_Length = u_length; return * this; }
  ct_Length &      operator = (unsigned long ul_length)
                     { ul_Length = ul_length; return * this; }
  unsigned         operator * (unsigned u)
                     { return u_Length * u; }
  unsigned long    operator * (unsigned long ul)
                     { return ul_Length * ul; }
  ....
  };
```

Wir haben die Mittel der Sprache C++ elegant angewendet und eine Lösung gefunden, in der keine einzige Typumwandlung erforderlich ist. Im arithmetischen Ausdruck `co_Length * sizeof (ct_Object *)` findet der Compiler automatisch, daß er für die Multiplikation die Methode `operator * (unsigned u)` der Klasse `ct_Length` anwenden muß. Wo liegt nun das Problem bei dieser Vorgehensweise? Ein Problem aus programmtechnischer Sicht gibt es nicht, aber ein Performance-Problem. Jede Klasse in C++ hat mindesten einen Konstruktor, Kopier-Konstruktor, Destruktor und Gleich-Operator. Wird er nicht explizit definiert, wie in der Klasse `ct_Length`, so werden diese Methoden vom Compiler automatisch erzeugt. Auch wenn wir einen gut optimierenden Compiler einsetzen, ist die Verwendung primitiver Datentypen effizienter. Die Typen für Längenangabe und EntryId werden sehr häufig genutzt. Würden wir sie als Klassen deklarieren, würde sich die Geschwindigkeit unseres Programms spürbar verlangsamen. Sehen wir uns dazu wieder ein Beispiel an.

```
ct_Length ct_Array:: GetLen () { return co_Length; }
....
ct_Array co_array;
ct_Length co_length3;
....
co_length3 = co_array. GetLen ();
```

In der letzten Programmzeile läuft nacheinander das Folgende ab: Aufruf der Methode `ct_Array:: GetLen ()`, Erzeugen eines temporären `ct_Length`-Objekts als Rückgabewert, Zuweisung des temporären Objekts an `co_length3` durch Aufruf der Methode `ct_Length:: operator = (const ct_Length &)` und Zerstören des temporären Objekts. Das Erzeugen des temporären Objekts besteht wiederum aus zwei Schritten: Speicher bereitstellen und Aufruf des Kopier-Konstruktors `ct_Length:: ct_Length (const ct_Length &)`. Temporäre Objekte werden nicht nur bei Rückgabewerten erzeugt sondern auch bei Methoden-Parametern, zum Beispiel in der Methode `ct_EntryId ct_Array:: Next (ct_EntryId co_id)`. Selbst wenn der Compiler gut optimiert, sind die eingebauten Mechanismen für Kopieren, Berechnen und Umwandeln einfacher Datentypen effizienter. Wir nehmen also einige manuelle Typumwandlungen in Kauf und bleiben bei den primitiven Datentypen für Längenangabe und EntryId. Zur Deklaration der Klasse `ct_Collection` fügen wir die beiden oben erörterten Typdefinitionen hinzu.

```
typedef unsigned long t_Length;
typedef unsigned long t_EntryId;

class ct_Collection: public ct_Object
  { ....
```

Die soeben gewonnenen Erkenntnisse werden wir im weiteren Verlauf des Buches wiederholt anwenden und fassen sie in einer Regel zusammen.

Bei Datentypen, die nur einen einzelnen Wert enthalten und sehr häufig verwendet werden, sind primitive Datentypen (`int`, `char` usw.) effizienter als Klassen.

Die EntryId, die eindeutige Identität jedes Eintrags in einer Collection, verwenden wir, um schneller auf die Elemente zugreifen zu können. Zwischen Array und DList gibt es aber einen wichtigen Unterschied bei der Verwendung von EntryIds. In einem Array ist die EntryId gleich dem Index im dynamischen Speicherblock. Wird im Array etwas hinzugefügt oder gelöscht, so verschieben sich alle dahinter stehenden Einträge, und es ändern sich die Indizes und EntryIds. Die EntryId eines einzelnen Elementes bleibt im allgemeinen nur solange gültig, wie im Array nichts geändert wurde. Abbildung 1-11 zeigt den Einfluß des Löschens auf die EntryIds in einem Array.

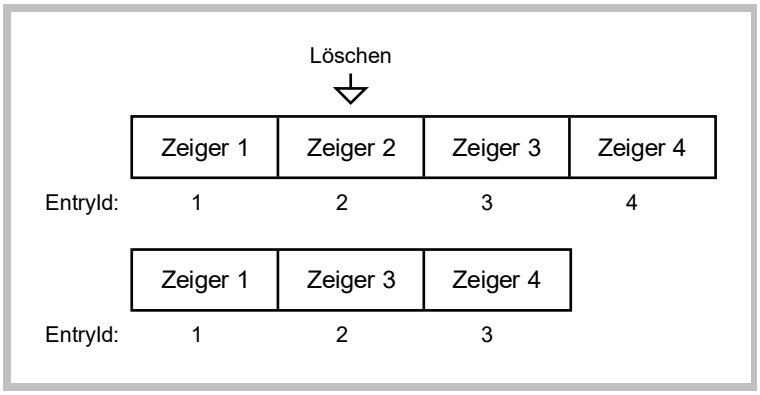

Abb. 1-11: *EntryIds beim Löschen in einem Array*

Bei der DList ist die EntryId gleich der Speicheradresse des Nodes, das den Zeiger auf das Objekt enthält. Das Node behält seine Adresse, auch wenn in der Liste Veränderungen vorgenommen werden. Wir können also ohne Bedenken EntryIds einer DList aufbewahren und später auf die damit referenzierten Objekte zugreifen. Die EntryId verliert erst ihre Gültigkeit, wenn das Element selbst aus der Collection entfernt wird. Sehen wir uns dazu Abbildung 1-12 an. Tabelle 1-1 faßt anschließend die wichtigsten Eigenschaften der beiden Collectionklassen `ct_Array` und `ct_DList` zusammen.

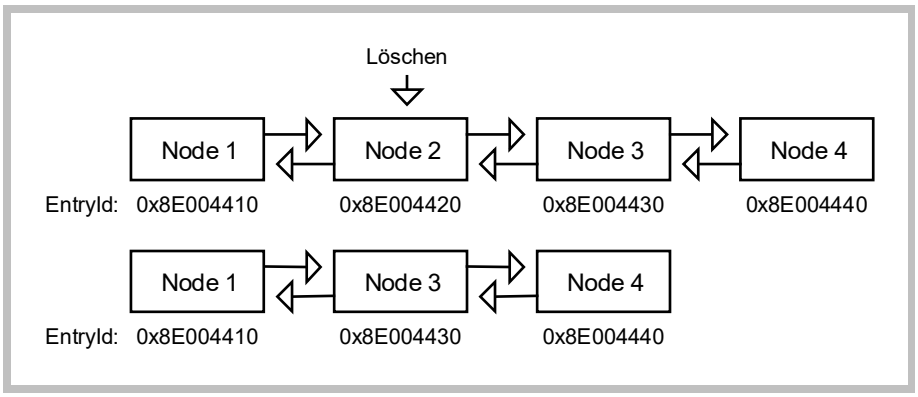

Abb. 1-12: *EntryIds beim Löschen in einerDList*

Eigenschaft	Array	DList
Speicherbedarf	wenig	viel
Anzahl Elemente	begrenzt	unbegrenzt
Gültigkeit EntryId	begrenzt	unbegrenzt
Verändern kleine Coll.	schnell	schnell
Verändern große Coll.	langsam	schnell

Tab. 1-1: *Wesentliche Eigenschaften der Collections*

1.4.4 Implementierung der Stringklasse

In der Standardbibliothek des C++-Compilers gibt es einen Modul für Zeichenkettenverarbeitung (string.h). Darin befinden sich Funktionen zum Kopieren, Anhängen und Bearbeiten. Es fehlen jedoch Funktionen zum Verwalten des Speichers der Zeichenkette und zum Einfügen und Löschen in einer Zeichenkette. Unsere eigene Klasse `ct_String` wird auf den Standardfunktionen aufbauen und die gewünschten Erweiterungen enthalten.

Die Länge der Zeichenkette soll dynamisch sein. Ähnlich wie beim Array benötigen wir einen Zeiger auf den dynamischen Speicherblock. Da sich in diesem Speicherblock eine Zeichenkette befindet, hat das Attribut die Definition `char * pc_Block`. In C und C++ sind Zeichenketten normalerweise nullterminiert. Unsere Stringklasse soll sich harmonisch in die vorhandene Umgebung standardisierter Bibliotheken einpassen. Wir übernehmen die Konvention und speichern deshalb am Ende des dynamischen Blocks ein Nullzeichen. Dabei ist zu beachten, daß das Nullzeichen bei der Berechnung der Länge der Zeichenkette nicht mitgezählt wird. Beim Anfordern von Speicher für den dynamischen Block muß es aber berücksichtigt werden. Die Standardfunktion `strlen` ermittelt die Länge einer nullterminierten Zeichenkette, indem sie den Speicher nach dem Nullzeichen durchsucht. Bei langen Zeichenketten kann das zu einem Rechenzeitproblem werden. Die Länge der Zeichenkette

wird häufig benötigt, zum Beispiel beim Einfügen oder Löschen von Teilzeichenketten (siehe Methode `Insert`). Deshalb nehmen wir in die Stringklasse ein zusätzliches Attribut `u_Length` auf, das zum schnelleren Zugriff die Länge enthält.

Auch die Klasse `ct_String` ist eine konkrete Klasse. Wir ergänzen die Methoden aus dem Design um Konstruktor, Kopier-Konstruktor, Destruktor und Gleich-Operator. Zur besseren Verarbeitung normaler Zeiger auf Zeichenketten (`char*`) nehmen wir einen weiteren Konstruktor und Gleich-Operator auf. Der folgende Programmausschnitt zeigt die vollständige Deklaration der Klasse `ct_String` und die Implementierung der Methode `Insert`.

```
class ct_String: public ct_Object
  {
  unsigned             u_Length;
  char *               pc_Block;
public:
                       ct_String ();
                       ct_String (const char * pc_init);
                       ct_String (const ct_String & co_init);
  virtual              ~ct_String ();
  ct_String &          operator = (const char * pc_asgn);
  ct_String &          operator = (const ct_String & co_asgn);
  virtual const char * GetTypeName () const;
  unsigned             GetLen () const;
  const char *         GetStr (unsigned u_pos = 0) const;
  void                 Insert (unsigned u_pos, const char * pc_ins);
  void                 Delete (unsigned u_pos, unsigned u_len);
  char &               operator [] (unsigned u_pos);
  };

void ct_String:: Insert (unsigned u_pos, const char * pc_ins)
  {
  ASSERT (u_pos <= u_Length);
  ASSERT (pc_ins != 0);
  unsigned u_inslen = strlen (pc_ins);
  if (u_inslen > 0)
    {
    u_Length += u_inslen;
    pc_Block = (char *) realloc (pc_Block, u_Length + 1);
    ASSERT (pc_Block != 0);
    memmove (pc_Block + u_pos + u_inslen, pc_Block + u_pos,
      u_Length - u_pos - u_inslen + 1);
    memcpy  (pc_Block + u_pos, pc_ins, u_inslen);
    }
  }
```

Mit der Klasse `ct_String` ist unsere Sammlung fundamentaler Klassen komplett. Wir haben uns bei der Implementierung redliche Mühe gegeben und sehen uns die kleine Klassenbibliothek noch einmal in Abbildung 1-13 an.

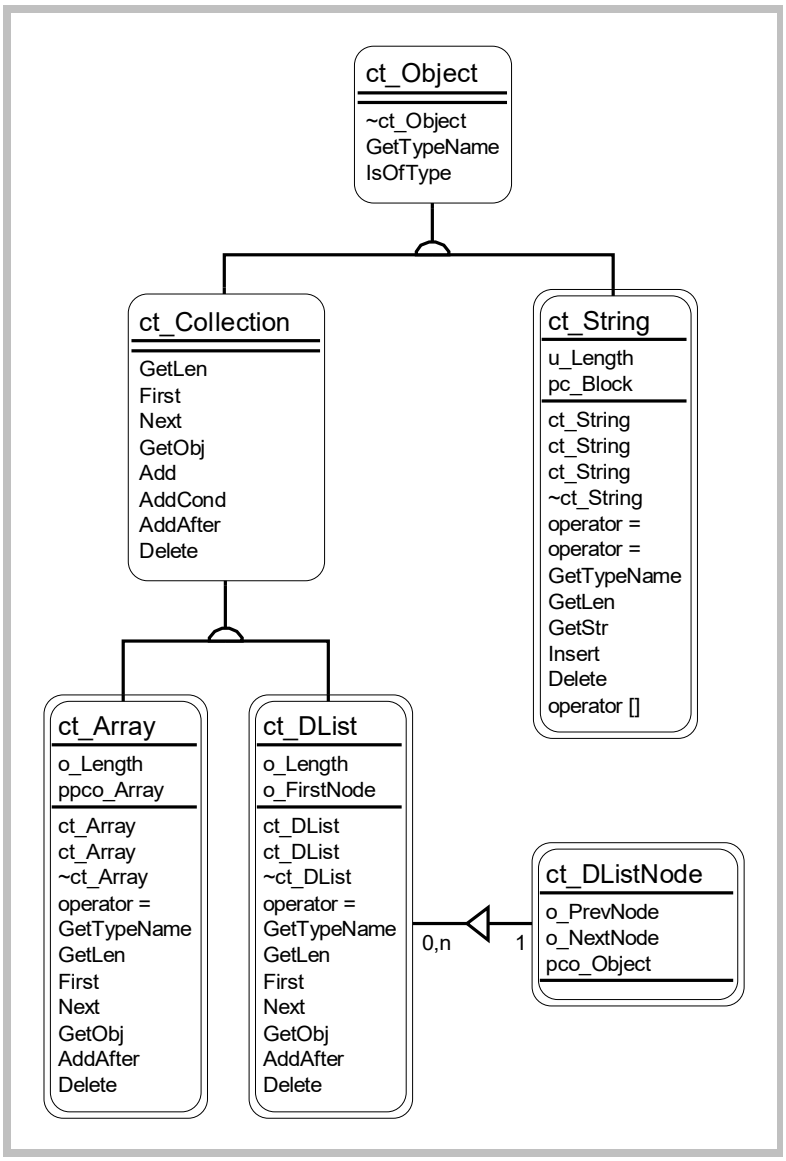

Abb. 1-13: *Die fundamentalen Klassen von OHelp*

1.5 Anwendungsklassen von OHelp

Unser Beispielprogramm OHelp ist das Informationsmodell (der Programmkern) eines interaktiven Hilfesystems. Die Anforderungen aus der Sicht des Anwenders des Hilfesystems wurden bereits aufgezählt und betreffen im wesentlichen die inhaltliche Seite des Informationsmodells.

Aus Sicht des Programmierers, der mit diesem Informationsmodell umgeht und eine Benutzeroberfläche entwirft, kommen eine Reihe konkreter Anforderungen hinzu. Diese Anforderungen betreffen vor allem die Programmierschnittstelle. Der Umgang mit dem Informationsmodell soll einfach sein. Änderungen interner Details sollen sich möglichst nicht auf die Schnittstelle auswirken. Im einzelnen bedeutet das:

- Der Zugriff auf Attribute erfolgt nicht direkt, sondern über Get- und Set-Methoden.
- Zeichenketten werden über normale Zeiger (`const char *`) ausgetauscht, nicht über die interne Stringklasse.
- Die häufigsten Typumwandlungen (Casts) sind vorprogrammiert.
- Themen können innerhalb eines Hypertextes kopiert werden.
- Die Konsistenz der verwalteten Daten muß sichergestellt sein.

1.5.1 Design der Anwendungsklassen

Ein Thema unseres Hilfesystems enthält einen Text. Ihm können mehrere Formatangaben und Hyperlinks zugeordnet werden. Zur internen Darstellung eines Textes mit Zusatzinformationen gibt es mehrere Möglichkeiten. Häufig modelliert man einen zeilenorientierten Text als Collection von Zeilen. Diese Form ist besonders für das Editieren großer Texte geeignet, erschwert aber die Positionierung der Zusatzinformationen. Bei Veränderungen im Text treten komplizierte Fallunterscheidungen auf, um Zeile und Spalte der Zusatzinformationen neu zu berechnen. Einfacher ist es, den Text als eine zusammenhängende Zeichenkette zu behandeln. Eine Textposition besteht nur noch aus einer einzelnen Angabe, der Spalte, und läßt sich bei Veränderungen leicht aktualisieren. Bei langen Zeichenketten können Einfüge- und Löschoperationen zu einem Rechenzeitproblem werden. In unserem Beispiel wird ein einzelner Text aber nicht sehr groß, denn er enthält genau eine Seite aus dem Hilfesystem. Deshalb entscheiden wir uns für die zweite Variante.

Ein Hyperlink ist ein Verweis von einem Thema zu einem anderen. Das Zielthema kann auf zwei Arten gespeichert werden, als Zeiger auf das Thema oder als EntryId aus der Collection der Themen. Wir geben der EntryId den Vorzug, denn damit gelangen wir sowohl zum Collectioneintrag als auch zum Thema. In der Liste der Forderungen an OHelp steht, daß ein Hyperlink optional eine Textposition besitzt. Unsere Klasse `ct_HyperLink` erhält also zwei Konstruktoren, einen mit der EntryId des Zielthemas und einen mit Position und EntryId. Wir können ein Hyperlink fragen, ob es eine Textposition besitzt (`IsInText`). Danach können wir die Position abfragen (`GetPos`) und verschieben (`MovePos`). Mit der Methode `GetTopicId` erhalten wir die EntryId des Zielthemas dieses Hyperlinks.

Eine Formatangabe stellen wir mit der Klasse `ct_Format` dar. Position, Länge und Textformat werden mit dem Konstruktor initialisiert. Die Position ist die Spalte im Text, ab der das Textformat gilt. Sie kann mit der Methode `GetPos` abgefragt und mit `MovePos` verschoben werden. Die Länge beschreibt den Gültigkeitsbereich des Textformats ab der Startposition. Auch dafür haben wir Methoden zum Abfragen (`GetLen`) und Ändern (`ChangeLen`). Das Textformat an sich codieren wir in einer ganzen Zahl mit hinreichend vielen Bits. Jedes Bit steht für ein einzelnes Textformat, zum Beispiel fett, kursiv, unterstrichen und Schreibmaschinentext. Durch logische Oder-Verknüpfung dieser Bits kann die Klasse `ct_Format` mehrere Textformate enthalten, ist jedoch unabhängig von den konkreten Textformaten. Die Klasse stellt Methoden zur Abfrage der Textformate (`GetFormat`), zum Hinzufügen (`AddFormat`) und zum Löschen (`DelFormat`) eines einzelnen Textformats bereit.

Überall dort, wo in einer Klasse eine Menge gleichartiger Objekte enthalten ist, nehmen wir die folgenden Methoden auf:

- Eine Methode zur Abfrage der gesamten Collection (z.B. `GetEntrys`).
- Eine Methode zur Abfrage eines einzelnen Objekts mit Typumwandlung (z.B. `GetEntry`).
- Eine oder mehrere Methoden zum Hinzufügen von Objekten (z.B. `AddEntry`).
- Eine Methode zum Löschen eines Objekts (z.B. `DelEntry`).

Mit der Methode `GetEntrys` können wir die gesamte Collection abfragen und durchlaufen. Sie liefert einen Zeiger vom Typ `const ct_Collection *`. Es können nur die `const`-Methoden der Collection aufgerufen werden, nicht die Methoden zum Einfügen und Löschen von Elementen. Beim Zugriff auf die Einträge erhalten wir von der Collection einen Zeiger auf die

Basisklasse ct_Object. Um die häufige Typprüfung mit der Methode IsOfType und die anschließende Typumwandlung zu sparen, nutzen wir die Methode GetEntry. Sie ermittelt aus der Collection den Zeiger auf das Objekt, prüft dessen Typ und gibt den umgewandelten Zeiger zurück.

Das Einfügen eines neuen Objekts besteht aus zwei Schritten, dem Erzeugen des Objekts und dem Einfügen in die Collection. Beides wird von der Methode AddEntry ausgeführt. Sie erhält die Konstruktor-Parameter des Objekts, erzeugt damit ein neues Objekt, fügt es in die Collection ein und gibt die neue EntryId zurück. Hat das Objekt mehrere Konstruktoren, gibt es auch mehrere überladene AddEntry-Methoden. Die Methode DelEntry löscht das referenzierte Objekt und entfernt den Eintrag aus der Collection. In der Collection wird nie direkt geändert, sondern nur indirekt über die Methoden AddEntry und DelEntry. Dadurch enthält die Collection stets gültige Zeiger. Alle erzeugten Objekte werden beim Entfernen aus der Collection gelöscht, und die Konsistenz der Collection ist gewährleistet.

Die Klasse zur Verwaltung eines Themas (ct_Topic) ist die umfangreichste unseres Beispielprogramms. Sie besitzt die größte Funktionalität. Ein Thema enthält zwei Mengen gleichartiger Objekte, die Formate und die Hyperlinks. Wir fügen in die Klasse ct_Topic die oben genannten Verwaltungsmethoden ein. Das sind für Hyperlinks die Methoden GetHyperLinks, GetHyperLink, AddHyperLink und DelHyperLink. AddHyperLink ist als überladene Methode doppelt vorhanden, denn die Klasse ct_HyperLink hat zwei verschiedene Konstruktoren. Zur Verwaltung der Formate dienen die Methoden GetFormats, GetFormat, AddFormat und DelFormat. AddHyperLink und AddFormat erzeugen nicht nur ein neues Element in der jeweiligen Collection, sondern sortieren es an der richtigen Stelle ein. Die Sortierung der Hyperlinks und Formate in aufsteigender Reihenfolge bezüglich der Textposition ist wichtig für den Programmteil, der die interne Darstellung auf dem Bildschirm anzeigen soll.

Für den Titel des Themas benötigen wir die Zugriffsmethoden GetTitle und SetTitle. Der Text wird als eine zusammenhängende Zeichenkette behandelt. Einen Zeiger darauf (const char *) erhalten wir mit der Methode GetText. Änderungen im Text können wir mit InsertText und DeleteText vornehmen. Eine Änderung im Text betrifft nicht nur die Zeichenkette, sonden auch die dahinter positionierten Zusatzinformationen. Die Methoden InsertText und DeleteText müssen also den Text und die Collections für Hyperlinks und Formate aktualisieren.

Die Klasse zur Darstellung des gesamten Hypertextes nennen wir ct_HyperText. Sie enthält Zugriffsmethoden für den Namen und das Wurzelthema (Get/SetName und Get/SetRootTopicId). Ein Hypertext enthält eine Menge Themen. Wir nehmen die zugehörigen Verwaltungsmethoden in die Klasse ct_HyperText auf (GetTopics, GetTopic, AddTopic und DelTopic). Zusätzlich war gefordert, daß ein Thema innerhalb eines Hypertextes kopiert werden kann. Dabei unterscheiden wir das Kopieren zu einem neuen Thema (CopyTopic) und das Überschreiben eines vorhandenen Themas (ReplaceTopic).

Mit diesen Überlegungen haben wir eine gute Grundlage für die Implementierung der Anwendungsklassen geschaffen. Abbildung 1-14 zeigt die wesentlichen Resultate unseres Designs.

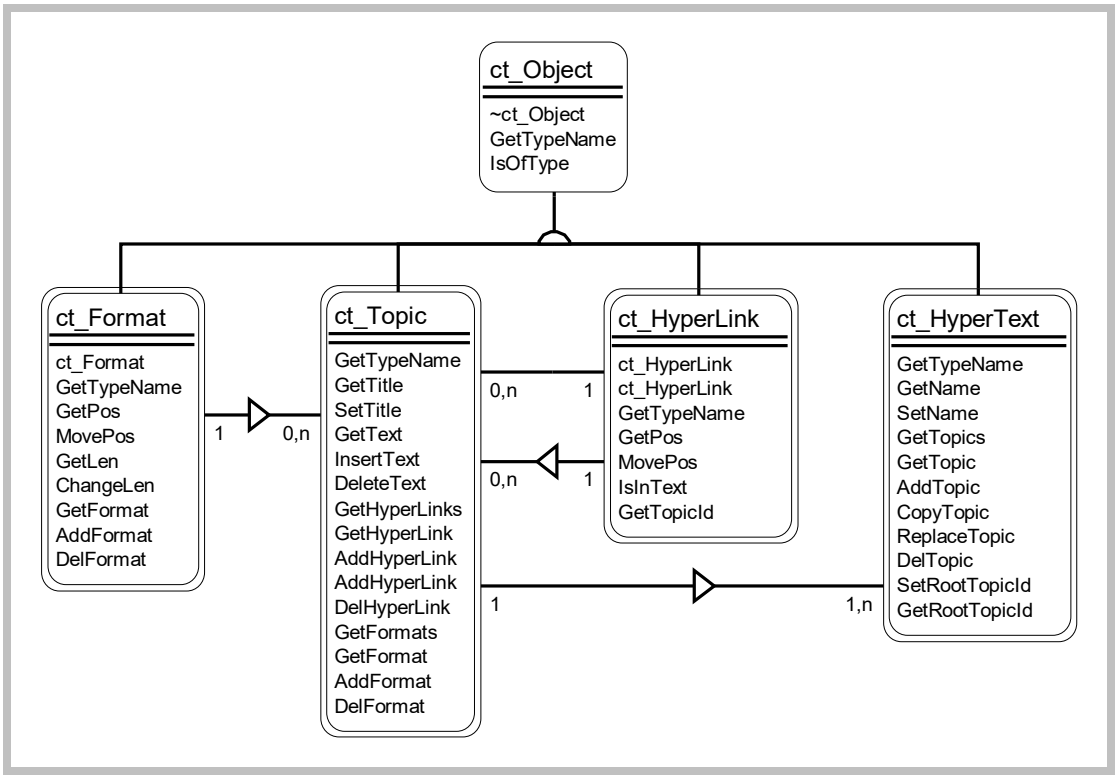

Abb. 1-14: *Design der Anwendungsklassen von OHelp*

1.5.2 Implementierung der Anwendungsklassen

Zur Implementierung der Klassen `ct_HyperLink` und `ct_Format` ist die wesentliche Arbeit bereits getan. Wir erweitern die Methodenschnittstelle aus dem Design um Parameter, Rückgabewerte und die Attribute. Beide Klassen enthalten ausschließlich primitive Datentypen und besitzen keine abhängigen Objekte. Deshalb funktionieren die vom Compiler automatisch generierten Methoden Kopier-Konstruktor, Destruktor und Gleich-Operator. Wir müssen sie nicht selbst definieren. Sehen wir uns nun die Deklaration beider Klassen und je eine Methodenimplementierung an.

```
class ct_HyperLink: public ct_Object
  {
  unsigned             u_Pos;
  bool                 b_InText;
  t_EntryId            o_TopicId;
public:
                       ct_HyperLink (t_EntryId o_id);
                       ct_HyperLink (unsigned u_pos, t_EntryId o_id);
  virtual const char * GetTypeName () const;
  unsigned             GetPos () const;
  void                 MovePos (int i_delta);
  bool                 IsInText () const;
  t_EntryId            GetTopicId () const;
  };

void ct_HyperLink:: MovePos (int i_delta)
  {
  ASSERT ((int) u_Pos + i_delta >= 0);
  u_Pos += i_delta;
  }
```

```
const unsigned cu_Bold      = 0x01;
const unsigned cu_Italic    = 0x02;
const unsigned cu_Underline = 0x04;
const unsigned cu_Example   = 0x08;

class ct_Format: public ct_Object
  {
  unsigned             u_Pos;
  unsigned             u_Len;
  unsigned             u_Format;
public:
                       ct_Format (unsigned u_pos, unsigned u_len,
                         unsigned u_format);
  virtual const char * GetTypeName () const;
  unsigned             GetPos () const;
  void                 MovePos (int i_delta);
  unsigned             GetLen () const;
  void                 ChangeLen (int i_delta);
  unsigned             GetFormat () const;
  void                 AddFormat (unsigned u_format);
  void                 DelFormat (unsigned u_format);
  };

void ct_Format:: DelFormat (unsigned u_format)
  {
  u_Format &= ~ u_format;
  }
```

Die Klasse `ct_Topic` enthält je eine Collection für Hyperlinks und Formate. Beide werden nur innerhalb der Klasse genutzt. Es existieren keine Verweise von außerhalb auf Elemente der Collections. Deshalb können wir zu deren Implementierung die speicherplatzsparende Collection `ct_Array` verwenden. In einem Objekt der Klasse `ct_HyperLink` ist kein Zeiger auf das referenzierte Thema enthalten, sondern die EntryId aus der Liste aller Themen. Um auf das Objekt zugreifen zu können, benötigen wir in der Klasse `ct_Topic` einen Verweis auf den Hypertext, denn der Hypertext enthält diese Liste der Themen. Der Zeiger auf den zugehörigen Hypertext wird im Konstruktor initialisiert und kann später nicht mehr geändert werden.

Ein Objekt der Klasse `ct_Topic` kann erst gelöscht werden, wenn keine Hyperlinks mehr auf dieses Objekt verweisen. Um das zu prüfen, müßten wir die Hyperlinks aller Themen durchlaufen. In derartigen Fällen verwendet man oft einen Referenzzähler. Das ist eine nichtnegative Zahl. Kommt ein neuer Verweis auf das Objekt hinzu, wird er um eins erhöht. Verschwindet ein Verweis, erniedrigt er sich um eins. Ist der Referenzzähler gleich Null, existieren keine Verweise mehr auf das Objekt, und es kann gelöscht werden. Wir fügen zur Klasse `ct_Topic` das Attribut `u_RefCount` und die Zugriffsmethode `GetRefCount` hinzu. Die Methoden `IncRefCount` und `DecRefCount` erhöhen bzw. erniedrigen den Referenzzähler um eins. Beide Methoden sind privat, denn der Referenzzähler wird nur von eigenen Methoden der Klasse `ct_Topic` geändert. Mit dem Referenzzähler haben wir gleichzeitig einen Beitrag zur Sicherung der Konsistenz geleistet. Zum Beispiel ist im Destruktor `~ct_Topic` die Anweisung `ASSERT (u_RefCount == 0)` enthalten.

In der Klasse `ct_Topic` haben der automatisch generierte Kopier-Konstruktor und Gleich-Operator nicht das gewünschte Verhalten. Beim Kopieren einer Collection (zum Beispiel mit `ct_Array:: operator =`) werden nur die Zeiger kopiert, nicht die referenzierten Objekte. Der automatisch generierte `ct_Topic:: operator =` nutzt den Gleich-Operator der Collection und würde die Hyperlinks und Formate nicht kopieren. Für die Klasse `ct_Topic` müssen wir also die Methoden Kopier-Konstruktor, Destruktor und Gleich-Operator selbst definieren. Das Leeren der Collections wird an zwei Stellen benötigt, beim Destruktor und beim Gleich-Operator. Wir nutzen dazu die private Methode `ClearCollections`. Ebenso gibt es eine private Methode zum Kopieren der Collections eines anderen Objekts der Klasse `ct_Topic`. Diese

Methode `CopyCollections` wird vom Kopier-Konstruktor und vom Gleich-Operator verwendet. Der folgende Programmausschnitt zeigt die vollständige Deklaration der Klasse `ct_Topic` und die Implementierung von `InsertText`. Zur Erinnerung sei noch einmal darauf hingewiesen, daß die Methode `InsertText` nicht nur den Text sondern auch die von der Änderung betroffenen Zusatzinformationen ändert.

```
class ct_Topic: public ct_Object
  {
  ct_HyperText *       pco_HyperText;
  unsigned             u_RefCount;
  ct_String            co_Title;
  ct_String            co_Text;
  ct_Array             co_HyperLinks;
  ct_Array             co_Formats;

  void                 IncRefCount ();
  void                 DecRefCount ();
  void                 ClearCollections ();
  void                 CopyCollections (const ct_Topic & co_copy);
public:
                       ct_Topic (ct_HyperText * pco_hyperText,
                         const char * pc_title);
                       ct_Topic (const ct_Topic & co_init);
  virtual              ~ct_Topic ();
  ct_Topic &           operator = (const ct_Topic & co_asgn);
  virtual const char * GetTypeName () const;
  ct_HyperText *       GetHyperText () const;
  unsigned             GetRefCount () const;
  const char *         GetTitle () const;
  void                 SetTitle (const char * pc_title);
  const char *         GetText () const;
  void                 InsertText (unsigned u_pos, const char * pc_ins);
  void                 DeleteText (unsigned u_pos, unsigned u_len);
  const ct_Collection * GetHyperLinks () const;
  ct_HyperLink *       GetHyperLink (t_EntryId o_hyperLinkId) const;
  t_EntryId            AddHyperLink (t_EntryId o_topicId);
  t_EntryId            AddHyperLink (unsigned u_pos, t_EntryId o_topicId);
  t_EntryId            DelHyperLink (t_EntryId o_hyperLinkId);
  const ct_Collection * GetFormats () const;
  ct_Format *          GetFormat (t_EntryId o_formatId) const;
  t_EntryId            AddFormat (unsigned u_pos, unsigned u_len,
                         unsigned u_format);
  t_EntryId            DelFormat (t_EntryId o_formatId);
  };

void ct_Topic:: InsertText (unsigned u_pos, const char * pc_ins)
  {
  t_EntryId o_id;
  unsigned u_insLen = strlen (pc_ins);
  co_Text. Insert (u_pos, pc_ins);
  for (o_id = co_HyperLinks. First ();
       o_id != 0;
       o_id = co_HyperLinks. Next (o_id))
    {
    ct_HyperLink * pco_hyli = GetHyperLink (o_id);
    if (pco_hyli-> GetPos () > u_pos)
      pco_hyli-> MovePos (u_insLen);
    }
  for (o_id = co_Formats. First ();
       o_id != 0;
       o_id = co_Formats. Next (o_id))
    {
    ct_Format * pco_format = GetFormat (o_id);
```

```
    if (pco_format-> GetPos () > u_pos)
      pco_format-> MovePos (u_insLen);
    else
      if (pco_format-> GetPos () + pco_format-> GetLen () > u_pos)
        pco_format-> ChangeLen (u_insLen);
    }
  }
```

Für die Klasse `ct_HyperText` wurde die wesentliche Arbeit bereits im Design getan. Wir erweitern die Methodenschnittstelle um Konstruktor und Destruktor. Das Kopieren eines vollständigen Hypertextes wird im Rahmen unseres Beispielprogramms nicht berücksichtigt. Der Hypertext enthält eine Collection von Themen. Es existieren Verweise auf die Einträge in dieser Collection, die Hyperlinks. Die Collection wird mit der Klasse `ct_DList` implementiert, denn in einer DList behalten die EntryIds auch nach Änderungen ihre Gültigkeit. Bevor ein Thema gelöscht werden kann, müssen alle Hyperlinks auf dieses Thema gelöscht werden. Dazu verwenden wir die Methode `DelTopicUsages`. Diese Methode wird von der Methode `DelTopic` aufgerufen, kann aber auch separat genutzt werden. Es folgt nun die Deklaration der Klasse `ct_HyperText` und die Implementierung der Methoden `DelTopicUsages` und `DelTopic`.

```
class ct_HyperText: public ct_Object
  {
  ct_String            co_Name;
  t_EntryId            o_RootTopicId;
  ct_DList             co_Topics;
public:
                       ct_HyperText ();
  virtual              ~ct_HyperText ();
  virtual const char * GetTypeName () const;
  const char *         GetName () const;
  void                 SetName (const char * pc_name);
  t_EntryId            GetRootTopicId () const;
  void                 SetRootTopicId (t_EntryId o_rootId);
  const ct_Collection * GetTopics () const;
  ct_Topic *           GetTopic (t_EntryId o_topicId);
  t_EntryId            AddTopic (const char * pc_title);
  t_EntryId            CopyTopic (t_EntryId o_source,
                         const char * pc_newTitle);
  void                 ReplaceTopic (t_EntryId o_repl, t_EntryId o_source,
                         const char * pc_newTitle);
  void                 DelTopicUsages (t_EntryId o_topicId);
  t_EntryId            DelTopic (t_EntryId o_topicId);
  };

void ct_HyperText:: DelTopicUsages (t_EntryId o_topicId)
  {
  for (t_EntryId o_id1 = co_Topics. First ();
       o_id1 != 0;
       o_id1 = co_Topics. Next (o_id1))
    {
    ct_Topic * pco_topic = GetTopic (o_id1);
    t_EntryId o_id2 = pco_topic-> GetHyperLinks ()-> First ();
    while (o_id2 != 0)
      if (pco_topic-> GetHyperLink (o_id2)-> GetTopicId () == o_topicId)
        o_id2 = pco_topic-> DelHyperLink (o_id2);
      else
        o_id2 = pco_topic-> GetHyperLinks ()-> Next (o_id2);
    }
  ASSERT (GetTopic (o_topicId)-> GetRefCount () == 0);
  }

t_EntryId ct_HyperText:: DelTopic (t_EntryId o_topicId)
  {
  if (o_topicId == o_RootTopicId)
```

```
    o_RootTopicId = 0;
  if (GetTopic (o_topicId)-> GetRefCount () > 0)
    DelTopicUsages (o_topicId);
  delete co_Topics. GetObj (o_topicId);
  return co_Topics. Delete (o_topicId);
  }
```

Nun ist unser Beispielprogramm OHelp komplett. Es ist sicher kein Ausgangspunkt für ein perfektes Hilfesystem. Das war auch nicht unsere Absicht. Wir wollten ein kleines, überschaubares Programm. Dieses Programm dient uns im folgenden als Studienobjekt für Performance-Analysen. Abbildung 1-15 zeigt die Implementierung der Anwendungsklassen von OHelp. Der Übersichtlichkeit halber enthält das Diagramm keine privaten Methoden.

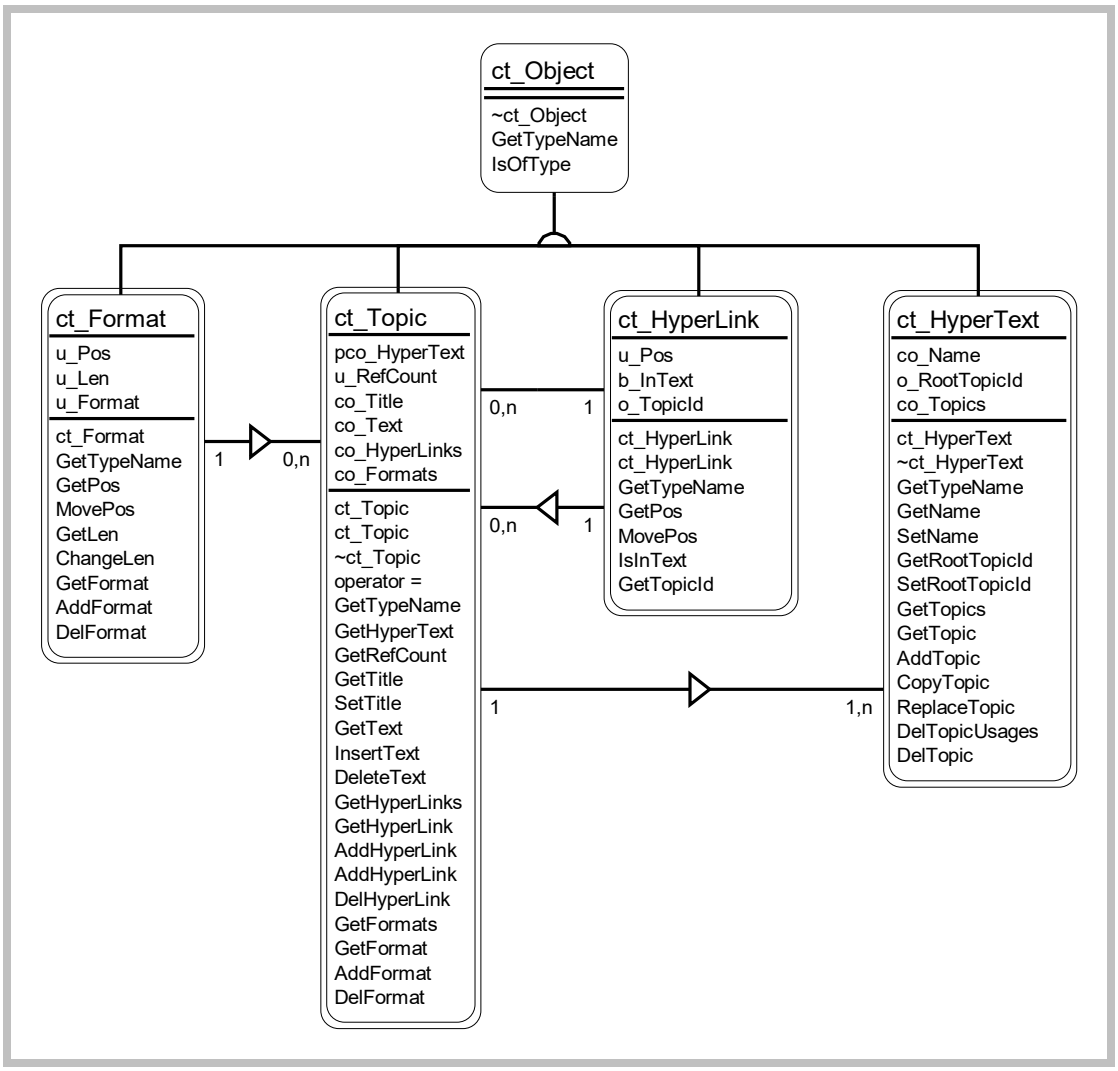

Abb. 1-15: *Die Anwendungsklassen von OHelp*

1.5.3 Stunde der Wahrheit

Bevor wir zu OHelp eine Benutzeroberfläche implementieren können, müssen wir alle Progammteile gründlich auf Herz und Nieren prüfen. Dazu schreiben wir ein Testprogramm. Darin werden alle Methoden mindestens einmal aufgerufen. Bei komplizierten Methoden, deren Verhalten vom Kontext abhängig ist, versuchen wir, jeden Kontext einmal zu durchlaufen. Auf diese Weise stellen wir die Richtiglkeit der Algorithmen sicher. Das ist aber nur die halbe Arbeit. Zu einem gut funktionierenden Programmteil gehört auch, daß die

Performance im Rahmen der Erwartungen bleibt. Die Performance (Betriebsverhalten, Leistungsfähigkeit) zeigt sich direkt in der Rechengeschwindigkeit und indirekt im Speicherbedarf. Bei einem zu hohen Speicherbedarf bremst die Verwaltung des virtuellen Speichers das Programm durch ständiges Ein- und Auslagern auf Festplatte, oder das Programm bricht wegen Speicherüberlaufs ab. Für einen kritischen Performancetest nutzen wir Datenmengen, wie sie in der Praxis im Grenzfall auftreten. Es ist keine Seltenheit, daß in einem Hilfesystem über 1000 Themen enthalten sind. Zum Testen von OHelp verwenden wir einem Hypertext mit folgendem Umfang:

- 10 000 Themen,
- Titel des Themas mit 15 Zeichen,
- pro Thema 10 Zeilen,
- pro Zeile 30 Zeichen Text und
- pro Zeile je eine Formatangabe und ein Hyperlink.

Nachdem OHelp den algorithmischen Teil des Tests erfolgreich bestanden hat, atmen wir erleichtert auf. Bei diesen kleinen Datenmengen verlief auch alles recht flott. Doch was passiert nun? Eigentlich sollte das Testprogramm den oben beschriebenen Hypertext aufbauen und darin einige Operationen ausführen. Stattdessen bleibt der Computer stehen und rührt sich nicht mehr. Haben wir wieder einmal vergessen, die Turbo-Taste zu drücken? Oder ist unser Programm in eine Endlosschleife geraten? Jetzt sehen wir wieder Bewegung am Computer. Die rote Kontrolldiode der Festplatte regt sich häufiger, und wir hören auch akustisch, daß die Festplatte immer stärker beansprucht wird. In unserem Testprogramm sind aber gar nicht so viele Dateioperationen enthalten. Es ist wohl das Betriebssystem, das mehr und mehr virtuellen Speicher ein- und auslagern muß. Wird das Programm auch die letzten Tests erfolgreich abschließen? Wir schwanken in unserer Hoffnung. Dann kommt das endgültige Aus. Auf dem Bildschirm steht groß und deutlich die Fehlermeldung *Out of memory*. Nachdem wir uns von diesem Schreck erholt haben, wiederholen wir die Performancetests mit geänderten Parametern. Dabei stellen wir nach und nach die folgenden Symptome an OHelp fest:

- Bei kleinen Datenmengen ist es flott und benötigt wenig Speicher.
- Beim Aufbau großer Datenmengen benötigt es viel Speicher.
- Reine Abfragen (ohne Veränderung) sind dann zu langsam.
- Beim Verändern großer Datenmengen wird es zunehmend langsamer.
- Im Laufe der Zeit benötigt es mehr und mehr Speicher, obwohl keine neuen Daten hinzukommen.

Bei der Entwicklung von OHelp haben wir nicht nur auf einen guten Programmierstil geachtet, sondern auch auf die Performance. Wir verwenden primitive Datentypen für Längenangabe und EntryId. Formatangaben und Hyperlinks werden mit der speicherplatzsparenden Collection `ct_Array` verwaltet. Ein nochmaliges Durchsehen der Programmtexte bringt keine offensichtlichen Effizienzfehler zutage. Was haben wir verkehrt gemacht? Woran liegt das schlechte Abschneiden im Performancetest?

Wer ein Haus baut, sollte Baugrund und Baumaterial genau prüfen. Und er sollte die Handwerker unter die Lupe nehmen, denen er den Bau anvertraut. Was im Baugewerbe üblich ist, darf auch ein Programmierer nicht außer acht lassen. Der Baugrund für OHelp ist die Programmiersprache C++. Das Baumaterial ist die C-Standardbibliothek. Die Handwerker sind der Compiler und der Linker. Die Programmiersprache haben wir bereits bei der Festlegung der Datentypen für Längenangabe und EntryId näher betrachtet. Dabei haben wir festgestellt, daß es in der Sprache C++ Konzepte gibt, die die Performance negativ beeinflussen. Unter diesem Blickwinkel prüfen wir in den folgenden drei Abschnitten auch den Compiler und die Standardbibliothek. Das Übersetzen virtueller und Inline-Methoden gehört in den Arbeitsbereich des Compilers. Die dynamische Speicherverwaltung ist Bestandteil der Standardbibliothek.

1.6 Ein Blick hinter die Kulissen des Compilers

1.6.1 Virtuelle Methoden

Betrachten wir *kurz* einige Begriffe, die mit virtuellen Methoden zusammenhängen. Vererbung und Polymorphie sind Basiskonzepte der objektorientierten Programmierung. Sie werden von jeder objektorientierten Sprache unterstützt. **Vererbung** bedeutet, daß eine Klasse ihre Eigenschaften auf eine andere Klasse überträgt, einschließlich der Attribute und Methoden. Man sagt, die geerbte Klasse ist die Basisklasse, und die erbende die abgeleitete Klasse. Das Konzept der Vererbung wird oftmals eingesetzt, ein Interface (eine Methodenschnittstelle) auf unterschiedliche Art zu implementieren. Das Interface wird durch eine abstrakte Basisklasse vorgegeben. Sie enthält nicht implementierte Methoden. Davon können keine Objekte gebildet werden, deshalb heißt sie abstrakt. In den abgeleiteten Klassen werden die Methoden implementiert. Eine Klasse, in der alle Methoden implemeniert sind, heißt konkrete Klasse, denn von ihr können Objekte erzeugt werden.

Ein Objekt einer abgeleiteten Klasse kann in einem Kontext verwendet werden, in dem nur das Interface der abstrakten Basisklasse bekannt ist. Dieses Konzept nennt man **Polymorphie**. Ein anschauliches Beispiel für Polymorphie ist in unserer kleinen Klassenbibliothek enthalten. Die abstrakte Basisklasse `ct_Collection` gibt das Interface für Collections vor. Die Klassen `ct_Array` und `ct_DList` sind konkrete Implementierungen davon. Über einen Zeiger können die Methoden der Klasse `ct_Collection` aufgerufen werden, ohne daß bekannt ist, welche konkrete Klasse sich hinter dem Zeiger verbirgt.

Wird eine Methode einer abstrakten Basisklasse aufgerufen, so entscheidet sich erst zur Laufzeit des Programms, welche konkrete Methode abgearbeitet werden muß. Dazu bedarf es einer neuen Technik, denn normalerweise sind dem Compiler die Methoden schon beim Übersetzen des Programms bekannt. Sehen wir uns ein Programmfragment an, in dem beide Fälle enthalten sind.

```
ct_String * pco_string = ....;
ct_Collection * pco_collection = ....;

pco_String-> GetLen ();     // Aufruf von ct_String:: GetLen ()
pco_collection-> GetLen (); // Nicht eindeutig,
  // Aufruf von ct_Array:: GetLen () oder ct_DList:: GetLen ()
```

Eine Methode, die in einem polymorphen Klassenbaum mehrmals implementiert wird, heißt **virtuelle Methode**. Die Methode `ct_String:: GetLen` ist es nicht. Virtuell sind hingegen `ct_Collection:: GetLen`, `ct_Array:: GetLen` und `ct_DList:: GetLen`. Die Methode `ct_Collection:: GetLen` ist **rein virtuell**, denn sie ist nicht implementiert. In einigen Programmiersprachen sind alle Methoden virtuell. Der Aufruf einer virtuellen Methode ist aber langsamer. Deshalb gibt es in C++ beide Formen, die nicht virtuelle und die virtuelle. Eine virtuelle Methode muß mit dem Schlüsselwort `virtual` gekennzeichnet werden. Beim Redefinieren einer virtuellen Methode in einer abgeleiteten Klasse ist das Schlüsselwort `virtual` optional.

Die technische Umsetzung virtueller Methoden ist nicht Bestandteil der Sprachdefinition von C++. Jeder Compiler kann dafür seine eigenen Mechanismen verwenden. Zum Ermitteln der aufzurufenden Methode muß jedes Objekt auf die konkreten Methoden verweisen. Da die Liste der konkreten Methoden für alle Objekte einer Klasse gleich ist, kann diese Liste wie ein statisches Attribut behandelt werden. Im Objekt selbst muß nur eine Klassen-Identität enthalten sein. Darüber kann aus der statischen Liste ein Verweis auf die richtige Methode ermittelt werden. Um den Zugriff zu beschleunigen, ist die Klassen-Identität meist ein direkter Zeiger auf die Liste, und in der Liste befinden sich direkte Zeiger auf die

Methoden. Zur Liste sagt man kurz **virtuelle Tabelle**. Zeiger und virtuelle Tabelle sind für den Programmierer unsichtbar. Wären sie sichtbar, würden sie für die Klasse `ct_Array` etwa so aussehen:

```
class ct_Array
  {
  void *         pv_ToVirtualTable;
  static void *  apv_VirtualTable [8];
  ....
  };

ct_Array:: ct_Array ()
  {
  pv_ToVirtualTable = & apv_VirtualTable;
  ....
  }

// Kein korrektes C++!
void * ct_Array:: apv_VirtualTable [8] =
  {
  & ct_Array:: ~ct_Array,
  & ct_Array:: GetTypeName,
  & ct_Array:: GetLen,
  & ct_Array:: First,
  & ct_Array:: Next,
  & ct_Array:: GetObj,
  & ct_Array:: AddAfter,
  & ct_Array:: Delete
  };
```

Der Zeiger auf die virtuelle Tabelle wird im Konstruktor initialisiert und später nicht mehr geändert, auch nicht durch einen Gleich-Operator. Die virtuelle Tabelle enthält Zeiger auf jede implementierte virtuelle Methode. Die Reihenfolge der Zeiger muß bei allen abgeleiteten Klassen gleich sein. Dann kann auf die Tabelle sehr schnell mit einem Index zugegriffen werden. Zum Beispiel muß in den virtuellen Tabellen der Klassen `ct_Array` und `ct_DList` der Destruktor den Index Null, `GetTypeName` den Index Eins, `GetLen` den Index Zwei usw. haben. Zum Aufruf der virtuellen Methode in `pco_collection-> GetLen ()` wird aus dem Objekt `*pco_collection` der Zeiger auf die virtuelle Tabelle ermittelt. Aus der Tabelle ergibt sich mit dem Index Zwei der Zeiger auf die konkrete virtuelle Methode. Schließlich wird diese Methode aufgerufen.

Aus dem Blickwinkel der Performance müssen wir bei der Arbeit mit virtuellen Methoden zwei Dinge beachten. Zum einen hat jede Klasse, in der eine virtuelle Methode enthalten ist, implizit ein zusätzliches Attribut. Das gilt auch für alle davon abgeleiteten Klassen. Zum anderen ist der Aufruf einer virtuellen Methode langsamer, denn er erfolgt über mehrfache Indirektion.

Aus der Verwendung virtueller Methoden resultiert nicht in jedem Fall eine Verschlechterung der Performance. In der Praxis kommt es auf das Verhältnis zwischen der Lösung mit und ohne virtuellen Methoden an. Besitzt eine Klasse schon hundert Attribute, spielt die Hinzunahme eines weiteren Attributs praktisch keine Rolle. Besitzt sie jedoch nur ein einzelnes Attribut vom Typ `char`, vergrößert sich der Umfang der Klasse durch die Verwendung virtueller Methoden um ein Vielfaches. Ähnlich verhält sich die Rechengeschwindigkeit. Die weiter oben dargestellte Methode `ct_Array:: AddAfter` enthält viele Anweisungen. Beim Aufruf wirkt sich die Indirektion nur unwesentlich auf die gesamte Abarbeitungszeit aus. Die Methode `ct_Array:: GetLen` hat hingegen nur die eine Anweisung `return o_Length`. Dabei ist der Unterschied zwischen einer nicht virtuellen und einer virtuellen Implementierung erheblich. Die folgende Regel faßt diese Beziehungen zusammen. In Abbildung 1-16 werden sie graphisch veranschaulicht.

Virtuelle Methoden können Rechenzeit und Speicherplatz belasten. An performancekritischen Stellen suchen wir eine Lösung ohne virtuelle Methoden. Existiert diese Lösung nicht, setzen wir weiterhin virtuelle Methoden ein.

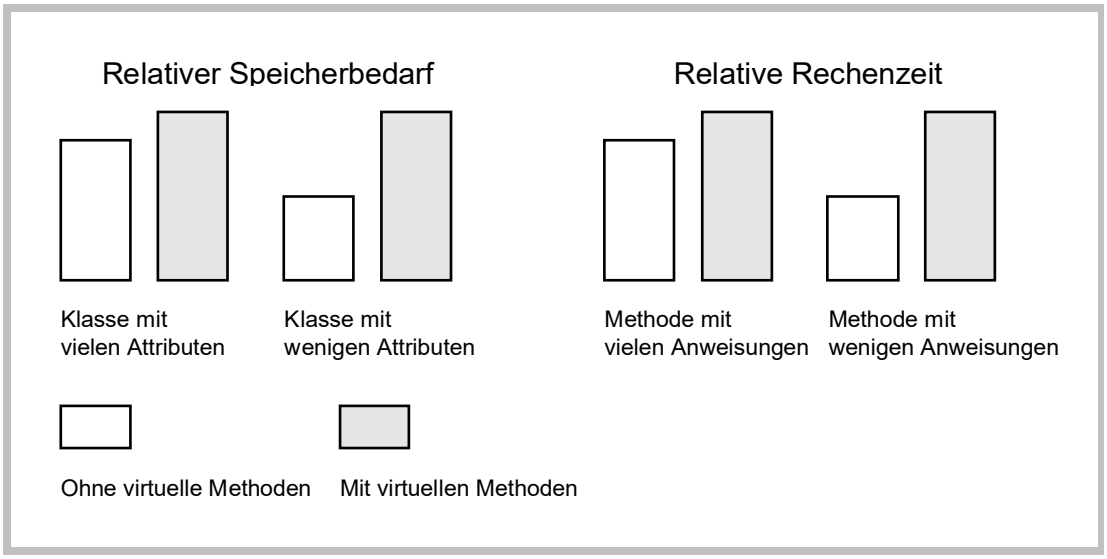

Abb. 1-16: *Einfluß virtueller Methoden auf die Performance*

1.6.2 Inline-Methoden

Das Konzept der Inline-Methoden ist kein Bestandteil der objektorientierten Programmierung. Es ist eine Erweiterung von C++ gegenüber C, um den Programmierstil zu verbessern. Was sind die Hintergründe für dieses Konzept? Einen Algorithmus, der mehrmals verwendet wird, faßt man in einer Methode zusammen. Eine Verwendung des Algorithmus ist gleich dem Aufruf der zugehörigen Methode. Beim Aufruf der Methode wird aber nicht nur der enthaltene Algorithmus abgearbeitet, sondern auch ein Methodenrahmen. Er ist für den Programmierer unsichtbar und wird vom Compiler automatisch hinzugefügt. Der Umfang des Methodenrahmens ist stark abhängig von Hardware, Betriebssystem, Compiler und Compilerschaltern. Vor dem Aufruf der Methode müssen interne Zustände der Hardware gesichert und nach dem Aufruf wiederhergestellt werden. Dieser Rahmen kostet natürlich Rechenzeit. Ähnlich wie bei den virtuellen Methoden ist auch die Auswirkung des Methodenrahmens auf die Performance von der Größe der Methode abhängig. Die Ausführungszeit einer Methode mit vielen Anweisungen wird durch den Methodenrahmen nur geringfügig vergrößert.

Es gibt in jedem Programm viele kleine Algorithmen, die nur aus einer einzelnen Anweisung bestehen. Wird ein kleiner Algorithmus als Methode implementiert und sehr häufig verwendet, kann durch den Methodenrahmen ein erheblicher Geschwindigkeitsverlust eintreten. In C hat man bei diesen Fällen den Präprozessor zu Hilfe genommen und für den Algorithmus ein Makro definiert. Das ist jedoch aus zwei Gründen kein guter Programmierstil. Zum einen sind Makros kein Bestandteil der Sprache C++. Durch Mischung von Makros und C++-Konstrukten verschlechtert sich die Lesbarkeit des Programms. Zum anderen wird die Fehlersuche erschwert. Beim fehlerhaften Aufruf einer Methode meldet der Compiler sehr genau Ursache und Textposition des Fehlers. Wird ein Makro falsch verwendet, beziehen sich Fehlermeldung und Textposition auf das expandierte Makro. Dieses ist aber im Programmtext nicht zu sehen, und die Rätselrunde zum Auffinden des Fehlers beginnt. Ein typisches Beispiel für einen kleinen Algorithmus, der häufig

verwendet wird, ist das Berechnen des Minimums zweier ganzer Zahlen. Sehen wir uns dazu ein Programmfragment an.

```
#define MIN(x, y) ((x) < (y) ? (x) : (y))
int i = MIN (5, -7);                    // Richtige Verwendung
char * pc = MIN ("Albert", "Andreas");  // Falsche Verwendung

inline int Min (int i1, int i2) { return i1 < i2 ? i1 : i2; }
int i = Min (5, -7);                    // Richtige Verwendung
char * pc = Min ("Albert", "Andreas");  // Falsche Verwendung
```

In der Definition der Methode `Min` haben beide Parameter einen Typ. Werden Parameter mit einem falschen Typ übergeben, meldet der Compiler sehr genau diesen Fehler. Das Makro `MIN` ist nicht typsicher, denn es ist eine reine Textoperation. Bei Verwendung des Makros ist aber die Ausführungsgeschwindigkeit deutlich besser. Im neuen Konzept der **Inline-Methode** sind beide Vorteile miteinander vereint. Eine Inline-Methode wird wie eine normale Methode deklariert und definiert und zusätzlich mit dem Schlüsselwort `inline` versehen. Eine Methode, deren Definition (Anweisungsteil) innerhalb der Klassendeklaration steht, gilt auch ohne Schlüsselwort als `inline`. Dadurch verschlechtert sich aber die Lesbarkeit des Programms. Definieren wir eine Inline-Methode außerhalb der Klassendeklaration, müssen wir das Schlüsselwort `inline` bei der Definition angeben. Optional kann es auch in der Deklaration stehen. Der Compiler kann eine Methode nur `inline` expandieren, wenn er ihre Definition gesehen hat. Deshalb sollte der Anweisungsteil unter der Klassendeklaration in der Headerdatei plaziert werden. Im folgenden Programmausschnitt sind beide Arten der Inline-Definition enthalten.

```
class ct_String: public ct_Object
  {
public:
  ....
  inline unsigned   GetLen () const;   // Gut lesbar
  const char *      GetStr (unsigned u_pos) const
    { ASSERT (u_pos <= u_Length);      // Anweisungsteil stört
      return pc_Block + u_pos; }       // an dieser Stelle
  ....
  };

// Inline-Definition in der Headerdatei plazieren!
inline unsigned ct_String:: GetLen () const
  {
  return u_Length;
  }
```

Die Verwendung einer Inline-Methode ist typsicher wie bei anderen Methoden. Der Compiler erzeugt beim Übersetzen aber keinen Methodenaufruf. Ähnlich wie der Präprozessor expandiert er die Methode direkt an der Verwendungsstelle. Dadurch entfällt der Methodenrahmen. Analog zum Makro ist eine Inline-Methode nur sinnvoll, wenn sie wenige Anweisungen enthält. Bei einer Methode mit vielen Anweisungen ist der Rechenzeitgewinn gering, und es tritt ein zusätzliches Problem auf. Zum Aufruf einer Methode erzeugt der Compiler nur wenig Code. Das Inline-Expandieren einer Methode mit vielen Anweisungen benötigt hingegen mehr Code. Wird die Methode häufig verwendet, vergrößert sich der Code des gesamten Programms spürbar. Sehen wir uns dazu Abbildung 1-17 an.

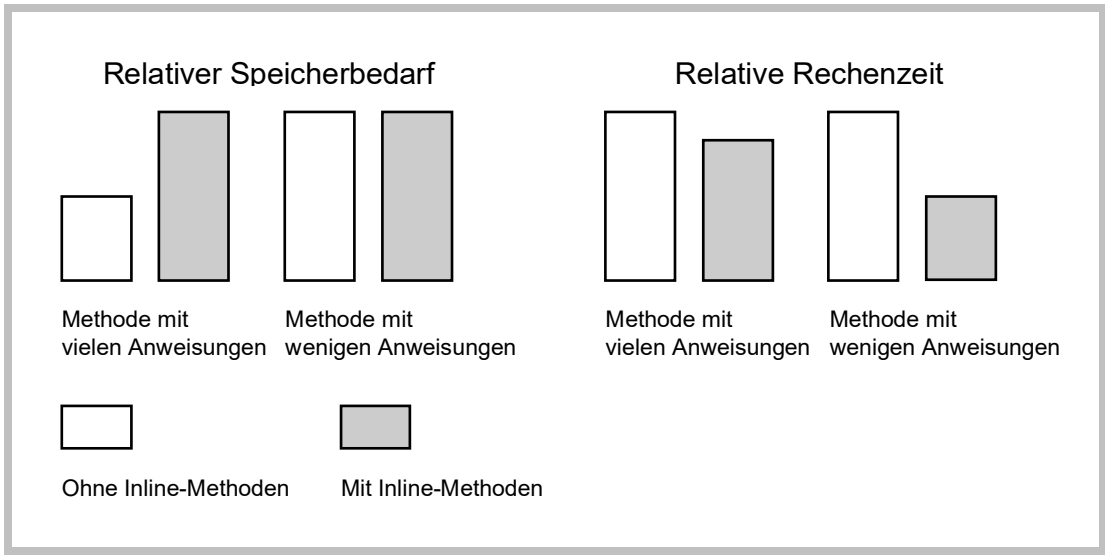

Abb. 1-17: *Einfluß von Inline-Methoden auf die Performance*

Die **Datenkapselung** ist ein Basiskonzept der objektorientierten Programmierung. Die Sprache C++ unterstützt dieses Konzept durch die Vergabe von Zugriffsrechten für Elemente einer Klasse. Attribute, Methoden usw. können `private`, `protected` oder `public` deklariert werden. In der Praxis sind Attribute meist privat, und der Zugriff erfogt indirekt über Methoden. Der Vorteil des indirekten Zugriffs ist, daß die Implementierung des Attributs dem Anwender der Klasse verborgen bleibt. Eine Änderung der internen Darstellung des Attributs betrifft nur selten die Zugriffsmethoden. Das Interface der Klasse bleibt dabei konstant. In den Anwendungsklassen von OHelp finden wir viele typische Zugriffsmethoden. Zum Beispiel wird der Titel eines Themas mit `GetTitle` abgefragt und mit `SetTitle` gesetzt. Beide Methoden arbeiten mit dem primitiven Datentyp `const char *`. Die interne Darstellung des Titels mit der Klasse `ct_String` ist für den Anwender nicht sichtbar. Würden wir später eine andere Stringklasse verwenden, bliebe das Interface der Zugriffsmethoden konstant, und der Anwender müßte nichts ändern.

Durch die Datenkapselung tauchen in einem C++-Programm viele kleine Methoden auf. Diese Zugriffsmethoden führen zu einer Verlangsamung des Programms. Erst mit Inline-Methoden wird der indirekte Zugriff auf Attribute effizient. Definieren wir Zugriffsmethoden stets `inline`, ist unsere objektorientierte Welt wieder in Ordnung. Wir können das Konzept der Datenkapselung einsetzen und müssen nicht auf die Performance verzichten.

Oder haben wir etwas übersehen?

Der Compiler kann eine Methode nur `inline` expandieren, wenn er genau weiß, um welche Methode es sich handelt. Beim Aufruf einer virtuellen Methode weiß er es im allgemeinen nicht. Ist eine Zugriffsmethode virtuell, müssen wir also einen doppelten Performanceverlust hinnehmen. Die konkrete Methode wird über mehrfache Indirektion ermittelt. Danach wird sie mit ihrem Methodenrahmen aufgerufen. Sehen wir uns unter diesem Aspekt noch einmal das Programmfragment mit virtuellen Zugriffsmethoden an.

```
ct_String * pco_string = ....;
ct_Collection * pco_collection = ....;

pco_String-> GetLen ();      // Aufruf von ct_String:: GetLen ()
pco_collection-> GetLen (); // Nicht eindeutig,
  // Aufruf von ct_Array:: GetLen () oder ct_DList:: GetLen ()
```

Ist die Zugriffsmethode `ct_String:: GetLen` eine Inline-Methode, ergibt sich daraus ein Rechenzeitgewinn. Das Inline-Definieren der Methoden `ct_Array:: GetLen` und `ct_DList:: GetLen` nützt uns in diesem Beispiel wenig, denn der Compiler weiß an der Verwendungsstelle

`pco_collection-> GetLen ()` nicht, welche Methode er `inline` expandieren soll. Stattdessen wird der Mechanismus zum Aufruf einer virtuellen Methode verwendet. Abbildung 1-18 verdeutlicht den doppelten Rechenzeitverlust, der in diesen Fällen eintritt.

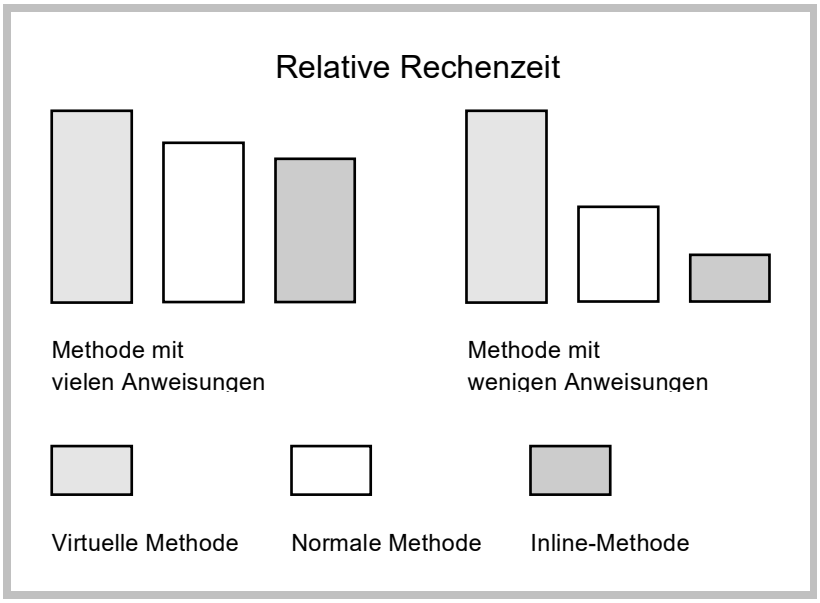

Abb. 1-18: *Rechenzeit von virtuellen, normalen und Inline-Methoden*

Zugriffsmethoden wie `GetLen` werden in einem Programm sehr häufig verwendet. Deshalb sollten wir uns überlegen, ob es sinnvoll ist, auch virtuelle Methoden `inline` zu definieren. Eine virtuelle Methode kann `inline` expandiert werden, wenn an der Verwendungsstelle die konkrete Klasse bekannt ist, zu der die Methode gehört. Es muß sichergestellt sein, daß das Objekt nicht zu einer abgeleiteten Klasse gehört, denn in der abgeleiteten Klasse könnte die Methode anders definiert sein. Der genaue Typ eines Objekts ist manchmal aus dem Kontrollfluß des Programms ersichtlich. Aber nicht alles, was der Programmierer sieht, sieht auch der Compiler. In diesen Fällen hängt es von der Qualität des Compilers ab, ob er die virtuelle Methode `inline` expandiert. Eindeutig ist die Methode erst, wenn der Programmierer die zugehörige Klasse mit dem ::-Operator angibt. Betrachten wir ein Programmfragment, in dem zahlreiche Verwendungen der virtuellen Methode `ct_Array:: GetLen` enthalten sind. Die Abkürzungen in den Kommentaren bedeuten:

- PO: Dem Programmierer ist der Typ des Objekts bekannt.
- PM: Dem Programmierer ist die konkrete Methode bekannt.
- CO: Dem Compiler ist der Typ des Objekts bekannt.
- CM: Dem Compiler ist die konkrete Methode bekannt.
- ?: Fraglich.
- ??: Sehr fraglich.
- ???: Überaus fraglich.

```
ct_Array co_array;
ct_Topic co_topic;
ct_Collection * pco_collection1 = new ct_Array;
const ct_Collection * pco_collection2 = co_topic. GetHyperLinks ();

// 1.
// Guter Programmierstil, aber langsam
co_array. GetLen ();         // PO PM CO   CM?
pco_collection1-> GetLen (); // PO PM CO?  CM??
pco_collection2-> GetLen (); // PO PM CO?? CM???

// 2.
// Weniger guter Programmierstil
```

```
co_array. GetLen ();                          // PO PM CO   CM?
((ct_Array *) pco_collection1)-> GetLen (); // PO PM CO?  CM??
((ct_Array *) pco_collection2)-> GetLen (); // PO PM CO?? CM???

// 3.
// Schnell, aber gefährlicher Programmierstil !!
co_array. ct_Array:: GetLen ();                          // PO PM CO   CM
((ct_Array *) pco_collection1)-> ct_Array:: GetLen (); // PO PM CO?  CM
((ct_Array *) pco_collection2)-> ct_Array:: GetLen (); // PO PM CO?? CM
```

Im ersten Teil geben wir dem Compiler keinerlei Hinweise. Beim Aufruf der Methode `GetLen` über das Objekt `co_array` ist der Typ des Objekts eindeutig. Der Compiler kann aber sagen: Nun, es ist eine virtuelle Methode, deshalb rufe ich sie auch wie eine virtuelle Methode auf. Im Falle des Zeigers `pco_collection1` kann der Compiler aus dem Kontrollfluß erkennen, daß diesem Zeiger ein Objekt der Klasse `ct_Array` zugewiesen wurde. Da der Zeiger seit der Initialisierung nicht geändert wurde, zeigt er immer noch auf ein Array. Beim Zeiger `pco_collection2` wird es schwieriger. Es muß berücksichtigt werden, daß die Methode `ct_Topic:: GetHyperLinks` einen Zeiger auf ein Objekt des Typs `ct_Array` liefert usw.

Im zweiten Teil geben wir dem Compiler den Hinweis, daß die Zeiger in beiden Fällen auf die abgeleitete Klasse `ct_Array` verweisen. Das reicht aber zum Ermitteln der konkreten Methode nicht aus, denn der Typ des referenzierten Objekts könnte von `ct_Array` abgeleitet sein. Der Compiler müßte wiederum Informationen aus dem Kontrollfluß zu Hilfe nehmen. Im dritten Teil beseitigen wir alle Unklarheiten durch genaue Angabe der Klasse, zu der die Methode gehört. Das ist in C++ die einzige Möglichkeit, eine virtuelle Methode nicht virtuell aufzurufen. In diesen Fällen *darf* der Compiler keinen Aufruf einer virtuellen Methode erzeugen.

Ist die Methode `ct_Array:: GetLen inline` definiert, haben wir im dritten Teil gute Chancen für einen Rechenzeitgewinn. Die genaue Angabe der Klasse mit dem ::-Operator erschwert aber die Programmpflege. Ändern wir zum Beispiel den Typ des Objekts `co_array` in `ct_MyArray`, müssen wir auch den Aufruf der Methode korrigieren und `co_array. ct_MyArray:: GetLen ()` schreiben. Die folgende Regel faßt diese Erkenntnisse zusammen.

Wir versuchen, performancekritische Zugriffsmethoden stets `inline` zu definieren. Wir erzwingen nicht das Inline-Expandieren mit dem ::-Operator. Stattdessen suchen wir eine Lösung ohne virtuelle Methoden.

1.6.3 Dynamische Speicherverwaltung

In der Entwicklungsphase eines Programms wissen wir nicht genau, wieviel Speicher es zur Laufzeit belegen wird. Zum Beispiel ist uns bei OHelp nicht bekannt, wieviele Themen in das Hilfesystem gelangen werden und wieviel Text jedes einzelne Thema enthalten wird. Zum Bereitstellen dieses Speichers benötigen wir eine **dynamische Speicherverwaltung**. Die Speicherverwaltung heißt dynamisch, weil Anzahl und Größe der angeforderten Speicherblöcke variieren können. Zur Verwaltung des Speichers gibt es in der C-Bibliothek des Compilers drei Standardfunktionen, `malloc` zum Anfordern, `realloc` zum Verändern der Größe und `free` zum Freigeben eines Blocks. Alle drei Funktionen arbeiten mit rohen Speicheradressen vom Typ `void *`. Bei einer Zeichenkette oder einem dynamischen Array kann sich die Größe des Speicherblocks im Laufe der Zeit ändern. In solchen Fällen verwendet man auch in C++ die C-Standardfunktionen.

Die Größe eines Objekts bleibt während seiner Lebensdauer konstant. Für das Erzeugen und Löschen von Objekten gibt es in C++ die komfortableren Operatoren `new` und `delete`. Der

Operator `new` fordert einen Speicherblock an, ruft den Konstruktor des Objekts auf und gibt einen typisierten Zeiger zurück.

Der Operator `delete` wird auf einen typisierten Zeiger angewendet. Er ruft den Destruktor des Objekts auf und gibt anschließend den belegten Speicher frei. Sehen wir uns als Beispiel eine Instanz der Klasse `ct_String` an. Die Größe des eigentlichen Objekts ist konstant. Es kann mit `new` erzeugt und mit `delete` gelöscht werden. Die Zeichenkette befindet sich jedoch in einem dynamischen Speicherblock und wird mit `malloc`, `realloc` und `free` verwaltet. Abbildung 1-19 zeigt das Speicherlayout einer String-Instanz. Jedes zusammenhängende Rechteck stellt einen einzelnen Speicherblock dar. Ist die Größe des Speicherblocks bekannt, markiert jeder kleine Strich an der linken Seite je ein Byte.

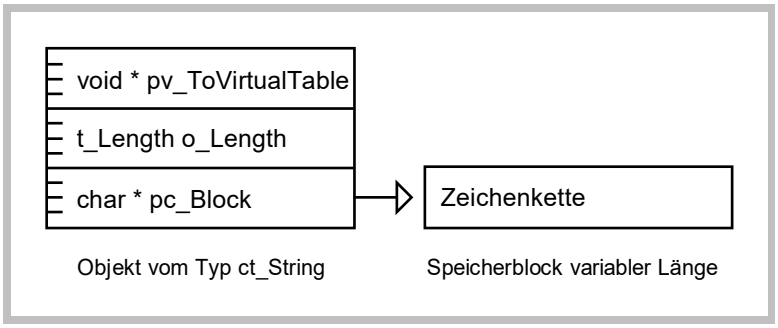

Abb. 1-19: *Speicherlayout einer Instanz der Klasse ct_String*

Jedes moderne Betriebssystem hat eine eigene Speicherverwaltung. Die Anwendungsschnittstelle unterscheidet sich aber von System zu System. Zum Beispiel arbeiten einige Betriebssysteme nicht mit Speicheradressen, sondern mit Handles. Die tatsächliche Speicheradresse erhalten wir erst nach Aufruf einer zusätzlichen Funktion. Die Speicherverwaltung eines Betriebssystems ist nicht auf kleine Anforderungen eingerichtet, wie wir sie zum Beispiel für ein String-Objekt benötigen. Deshalb ist in der C-Standardbibliothek eine eigene Speicherverwaltung enthalten. Sie fordert vom Betriebssystem große Blöcke an und gibt kleinere an das Anwendungsprogramm weiter. Die Schnittstelle dieser Speicherverwaltung ist vom Betriebssystem unabhängig und besteht aus den genannten Funktionen `malloc`, `realloc` und `free`. In Abbildung 1-20 sind typische Speicheranforderungen eines Anwendungsprogramms und der C-Standardbibliothek zu sehen.

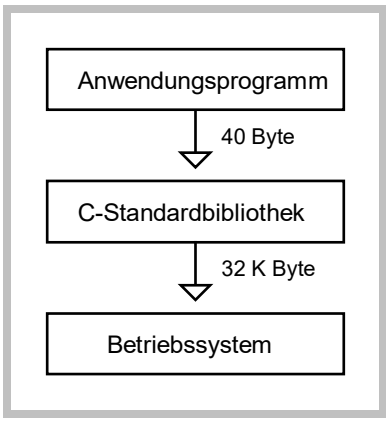

Abb. 1-20: *Typische Speicheranforderungen*

Um die Speicherverwaltung effektiv für unsere Zwecke einsetzen zu können, müssen wir uns näher mit ihrer Funktionsweise befassen. Betrachten wir zunächst die genaue Deklaration der drei Standardfunktionen.

```
void * malloc (unsigned u_size);
void * realloc (void * pv_block, unsigned u_newsize);
void free (void * pv_block);
```

Die Funktion `malloc` erhält als Parameter die Größe des bereitzustellenden Speichers. Sie liefert einen Zeiger auf einen Block, der mindestens die angegebene Größe hat. An die Funktion `free` wird nur ein Zeiger auf den Block übergeben. Wie groß der freizugebende Block ist, muß die Speicherverwaltung selbst wissen. Das ist eine Erleichterung für den Anwender. Er muß sich die Größe des angeforderten Speichers nicht merken. Für die Speicherverwaltung bedeutet es aber einen Zusatzaufwand. Einem Aufruf von `realloc` entspricht je einem Aufruf von `malloc` und `free`, wobei der Inhalt des in der Größe veränderten Speicherblocks erhalten bleiben muß.

Die Freigabe eines Blocks darf die belegten Blöcke nicht beeinflussen, denn das Anwendungsprogramm setzt voraus, daß ein angeforderter Speicherblock seine Adresse behält. Durch den freigewordenen Block entsteht also im Speicher ein ungenutzter Bereich. Auch die Freiblöcke müssen verwaltet werden. Beim Anfordern versucht die Speicherverwaltung zunächst, die vorhandenen Freiblöcke zu füllen. Gelingt es nicht, muß vom Betriebssystem neuer Speicher geholt werden. Durch wiederholtes Anfordern und Freigeben des Speichers entsteht eine Kette von Freiblöcken, die den Speicher in kleine Teile zerlegt. Dies nennt man **Speicherfragmentierung**. Um eine Fragmentierung innerhalb der Freiblöcke zu vermeiden, wird bei jeder Freigabe geprüft, ob der freigewordene Block physisch an andere freie Blöcke grenzt. In diesem Fall werden benachbarte Freiblöcke zu einem größeren Block zusammengefaßt. Durch Rundung der Größe der Blöcke wird der Speicherfragmentierung entgegengewirkt. Typischerweise wird jede Anforderung des Anwendungsprogramms auf die nächsthöhere 8- oder 16-Byte-Grenze gerundet. Werden zum Beispiel 18 Bytes freigegeben und anschließend 22 Bytes angefordert, kann der freigewordene Block genutzt werden. Pro Speicherblock (frei oder belegt) entsteht der nachstehende Verwaltungsaufwand:

- Speichern der Größe des Blocks,
- Speichern von Informationen über benachbarte Blöcke und
- Rundung der Größe.

Wie groß dieser Aufwand tatsächlich ist, hängt von der Implementierung der Speicherverwaltung ab. Eine effektive Codierung der Zusatzinformationen pro Block könnte etwa so aussehen: 15 Bit für die Länge (8 Bytes bis 256 KB in 8-Byte-Schritten); ein Bit für die Information, ob der Block belegt oder frei ist; 15 Bit für die Länge des Vorgänger-Blocks. Die Adresse des Vorgängers ergibt sich aus der Adresse des Blocks vermindert um die Länge des Vorgängers. Die Adresse des Nachfolgers entsteht durch Addition der eigenen Adresse mit der eigenen Länge. Die Summe dieser Informationen beträgt 31 Bit (4 Bytes).

Wegen der Rundung auf die nächsthöhere 8-Byte-Grenze bleiben null bis sieben Bytes ungenutzt. Pro Speicherblock müssen wir also vier bis elf Bytes für die Verwaltung einkalkulieren. Das fällt insbesondere bei kleinen Anforderungen ins Gewicht. Hinzu kommt die im Laufe der Zeit wachsende Fragmentierung. Diese führt nicht nur zu brach liegendem Speicher, sondern auch zu einer Verlangsamung der Verwaltung. Bei jeder neuen Anforderung muß die Freiliste geprüft werden. Je länger sie wird, desto länger dauert im Durchschnitt diese Überprüfung. Es gibt verschiedene Algorithmen zum Optimieren der Freiliste. Zum Beispiel ist es üblich, daß sich der freie Speicher selbst verwaltet und keinen zusätzlichen Speicher belegt. Dennoch hat jede dynamische Speicherverwaltung folgende Nachteile:

- Sie ist ungeeignet für kleine Anforderungen von zehn oder zwanzig Bytes.
- Häufiges Anfordern und Freigeben führt zur Fragmentierung.
- Sie ist ungeeignet für viele Anforderungen derselben Größe.

Der zuletzt genannte Nachteil macht sich auch in unserem Hilfesystem bemerkbar. Darin sind viele kleine Objekte vom Typ `ct_HyperLink` und `ct_Format` enthalten. In einer statischen Speicherverwaltung, die speziell auf Objekte dieser Größe ausgerichtet ist, könnte der Verwaltungsaufwand pro Objekt auf fast Null reduziert werden. Eine dynamische Speicherverwaltung kennt solche Optimierungen jedoch nicht. Alle Nachteile sind für den Programmierer unsichtbar. Sie verstecken sich hinter den Standardfunktionen `malloc`, `realloc` und `free` und hinter den Operatoren `new` und `delete`. Wie bei den virtuellen und den Inline-Methoden können wir auch nach diesem Blick hinter die Kulissen des Compilers die Schwachstellen unseres Programms leichter aufdecken.

1.7 Performance-Analyse von OHelp

Alle bisherigen Erfahrungen zusammengenommen, können wir nun den folgenden Fragekatalog aufstellen. Zur Abkürzung bedeutet "Kleine Methode" Methode mit wenigen Anweisungen und "Kleine Klasse" Klasse mit wenigen Attributen. Die Antwort Ja steht in jedem Fall für eine schlechte Performance.

1. Gibt es Klassen, die nur einen einzelnen Wert enthalten und sehr häufig verwendet werden?
2. Gibt es kleine Methoden, die sehr häufig aufgerufen werden, aber nicht `inline` definiert sind?
3. Gibt es kleine virtuelle Methoden, die sehr häufig aufgerufen werden?
4. Gibt es kleine Klassen, die sehr häufig verwendet werden und einen virtuellen Tabellenzeiger enthalten?
5. Treten bei einer typischen Datenmenge sehr viele kleine Speicherblöcke auf?
6. Gibt es große Mengen Speicherblöcke derselben Größe?

Die erste Frage können wir zu unseren Gunsten mit Nein beantworten. Alle anderen Fragen müssen wir bejahen. Die genaue Analye dieses schlechten Abschneidens wird in den folgenden beiden Abschnitten vorgenommen. Die Rechengeschwindigkeit läßt sich nur relativ bestimmen. Der Speicherbedarf wird hingegen mit konkreten Datenmengen genau berechnet.

1.7.1 Rechenzeitverhalten

Bei der Implementierung von OHelp haben wir keine Inline-Methoden verwendet. Wäre die Rechengeschwindigkeit zufriedenstellend, könnten wir auch dabei bleiben. Im Performancetest mit großen Datenmengen war das Programm aber zu langsam. Der Zugriff auf die Attribute einer Klasse kann fast immer mit einer Inline-Methode erfolgen. Es können jedoch auch andere Methoden, die nur aus ein oder zwei Anweisungen bestehen, `inline` definiert werden. Das Umwandeln in eine Inline-Methode ist schnell erledigt. Wir verwenden das Schlüsselwort `inline` und verschieben die Definition in die Headerdatei. Sehen wir uns die so geänderte Deklaration von `ct_Topic` an.

```
class ct_Topic: public ct_Object
  {
  ....
  inline void                IncRefCount ();
  inline void                DecRefCount ();
public:
  inline virtual const char * GetTypeName () const;
  inline ct_HyperText *       GetHyperText () const;
  inline unsigned             GetRefCount () const;
```

```
  ....
  };

inline void ct_Topic:: IncRefCount ()
  {
  u_RefCount ++;
  }

inline void ct_Topic:: DecRefCount ()
  {
  ASSERT (u_RefCount > 0);
  u_RefCount --;
  }
```

Das Rechenzeitverhalten eines Programms wird im wesentlichen durch die zeitkritischen Stellen bestimmt. Zeitkritisch sind die Anweisungen, die am häufigsten abgearbeitet werden. Am genauesten finden wir diese Stellen mit einem Programmierwerkzeug, dem Profiler. Der Profiler untersucht das Programm zur Laufzeit und liefert eine Statistik. Die Aussagekraft dieser Statistik hängt stark vom jeweiligen Profiler und davon ab, wie lange wir uns mit ihm beschäftigen. Der Profiler hilft uns erst zu einem Zeitpunkt, an dem es eigentlich schon zu spät ist. Das nachträgliche Ändern getesteter Programmteile ist stets mit Risiken verbunden. Besser ist das frühzeitige Aufdecken der zeitkritischen Stellen in der Entstehungsphase des Programms. Beim Design können wir diese Stellen grob einkreisen. Während der Implementierung achten wir besonders auf Schleifen im Anweisungsteil. Treten geschachtelte Schleifen auf, sind die inneren Schleifen am meisten zeitkritisch. Sehen wir uns unter diesem Aspekt ein Programmfragment an. Darin sind eine Methode der Klasse `ct_Topic` und ein Methodenteil der Benutzerschnittstelle zum formatierten Anzeigen eines Themas enthalten. Der Kommentar "1 VM" bedeutet, daß in dieser Programmzeile eine virtuelle Methode aufgerufen wird. Virtuelle Methoden innerhalb eines `ASSERT`-Makros zählen nicht.

```
ct_Format * ct_Topic:: GetFormat (t_EntryId o_formatId) const
  {
  ct_Object * pco_obj = co_Formats. GetObj (o_formatId); // 1 VM
  ASSERT (pco_obj-> IsOfType ("ct_Format"));      // Zählt nicht
  return (ct_Format *) pco_obj;
  }

void ct_TopicView:: ViewFormatted (ct_Topic * pco_topic)
  {
  ct_String co_str = pco_topic-> GetText ();
  const ct_Collection * pco_formats = pco_topic-> GetFormats ();
  t_EntryId o_id;
  for (o_id = pco_formats-> First ();                      // 1 VM
       o_id != 0;
       o_id = pco_formats-> Next (o_id))                   // 1 VM
    {
    ct_Format * pco_format = pco_topic-> GetFormat (o_id); // 1 VM
    ....
    }
  const ct_Collection * pco_hyperLinks = pco_topic-> GetHyperLinks ();
  for (o_id = pco_hyperLinks-> First ();                       // 1 VM
       o_id != 0;
       o_id = pco_hyperLinks-> Next (o_id))                    // 1 VM
    {
    ct_HyperLink * pco_hyli = pco_topic-> GetHyperLink (o_id); // 1 VM
    ....
    }
  }
```

Der Aufruf von `ct_Array:: GetObj` in `ct_Topic:: GetFormat` kann optimiert werden, indem wir auch in der Klasse `ct_Array` kleine Methoden `inline` definieren. Dann liegt es aber noch am

Compiler, ob er die Methode an dieser Verwendungsstelle tatsächlich `inline` expandiert. Die anderen virtuellen Methoden lassen sich nur durch genaue Angabe des Objekttyps eliminieren. Das ist aber nicht im Sinne der Polymorphie.

Nicht zufällig gehören in diesem Programmfragment alle virtuellen Methoden zu den Collections. Zeitkritische Stellen sind innerhalb von Schleifen. Wie entsteht eine Schleife? Dies geschieht durch Iterieren einer Collection. Das bedeutet, daß an fast allen zeitkritischen Stellen eines Programms die Collections beteiligt sind. Den Aspekt Geschwindigkeits-Optimierung haben wir bei der Entwicklung unserer Collections nicht berücksichtigt. Eine kleine nachträgliche Änderung wie bei den Inline-Methoden ist in diesem Fall nicht möglich. Wir benötigten ein grundlegend anderes Konzept für Collections. Die folgende Regel faßt diese Erkenntnisse zusammen.

An fast allen zeitkritischen Stellen eines Programms werden Collections verwendet. Collections mit virtuellen Methoden sind für ein schnelles Iterieren und Zugreifen ungeeignet.

Eine Collection mit virtuellen Methoden ist vergleichbar mit einem Flugzeug, das aus leichtem Holz oder Kunststoff besteht. Für den Transport kleiner Güter und für einen Rundflug über einer Stadt reicht es aus. Wollen wir aber viele Güter transportieren und große Stecken überwinden, benötigen wir ein Düsenflugzeug aus einer stabilen Metallkonstruktion.

1.7.2 Speicherbedarf

Für jede einzelne Klasse von OHelp untersuchen wir die folgenden Kriterien: Absolute Größe in Bytes, Anteil virtueller Tabellenzeiger und Anzahl der Speicherblöcke. Bei komplexen Objekten betrachten wir das eigentliche Objekt und die abhängigen Objekte getrennt. Zur Berechnung der absoluten Größe von Objekten setzen wir einen 32-Bit-Compiler voraus. Die primitiven Datentypen umfassen: `char` ein Byte, `short` zwei Bytes, `int` vier Bytes, `long` vier Bytes und Zeiger vier Bytes.

Wir beginnen mit den fundamentalen Klassen. Weiter oben sahen wir bereits das Speicherlayout eines Strings. Betrachten wir nun die Klasse `ct_Array`. Bei Design und Implementierung haben wir uns besondere Mühe gegeben, diese Collection speicherplatzoptimal zu gestalten. Abbildung 1-21 zeigt das Speicherlayout eines Arrays mit n Einträgen. Die genauen Analyseergebnisse befinden sich in Tabelle 1-2. Die in der Collection enthaltenen Objekte sind kein Bestandteil der Collection. Wir müssen aber beachten, daß jedes dieser Objekte einen eigenen Speicherblock erfordert.

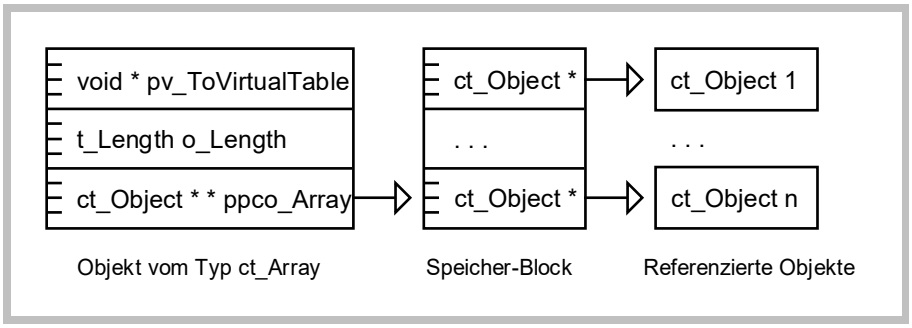

Abb. 1-21: *Speicherlayout einer Instanz der Klasse ct_Array*

Eine DList ist komfortabler als ein Array. Diesen Komfort müssen wir mit erhöhtem Speicherbedarf bezahlen. Jeder Eintrag erfordert ein Node. Dieses Node belegt nicht nur

Speicher. Es wird auch in einem einzelnen Block untergebracht. In Abbildung 1-22 sehen wir das Speicherlayout einer Collection mit n Einträgen.

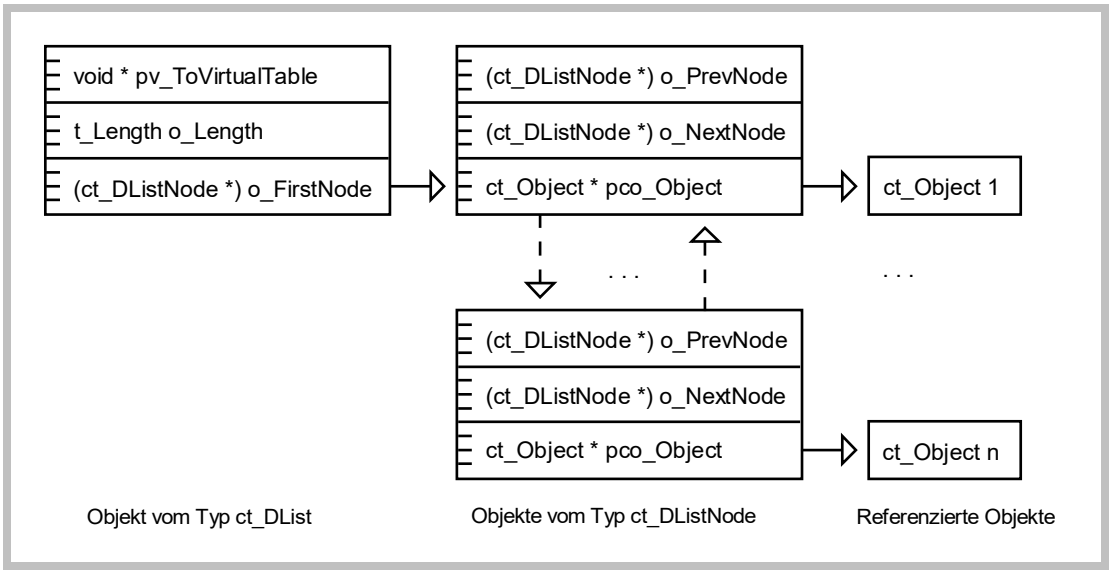

Abb. 1-22: *Speicherlayout einer Instanz der Klasse ct_DList*

Objekttyp	Absolute Größe	Virt. Tab.-Zeiger	Anzahl Blöcke
String ohne Inhalt	12 Bytes	4 Bytes	1
Zeichenkette der Länge n	n + 1 Bytes	0 Bytes	1
String der Länge n	12 + n + 1 Bytes	4 Bytes	2
Array ohne Inhalt	12 Bytes	4 Bytes	1
1 Array-Eintrag	4 Bytes	0 Bytes	0
Array mit n Einträgen	12 + 4 * n Bytes	4 Bytes	2
DList ohne Inhalt	12 Bytes	4 Bytes	1
1 DList-Eintrag	12 Bytes	0 Bytes	1
DList mit n Einträgen	12 + 12 * n Bytes	4 Bytes	1 + n

Tab. 1-2: *Speicheranalyse für String, Array und DList*

Für eine detaillierte Analyse des Speicherbedarfs der Anwendungsklassen nutzen wir dieselben Daten wie beim ersten Performancetest. Zur Erinnerung seien diese Zahlen noch einmal wiederholt. In Abbildung 1-23 sehen wir das Speicherlayout eines einzelnen Themas. Tabelle 1-3 enthält die Analyseergebnisse.

- 10 000 Themen,
- Titel des Themas mit 15 Zeichen,
- pro Thema 10 Zeilen,
- pro Zeile 30 Zeichen Text und
- pro Zeile je eine Formatangabe und ein Hyperlink.

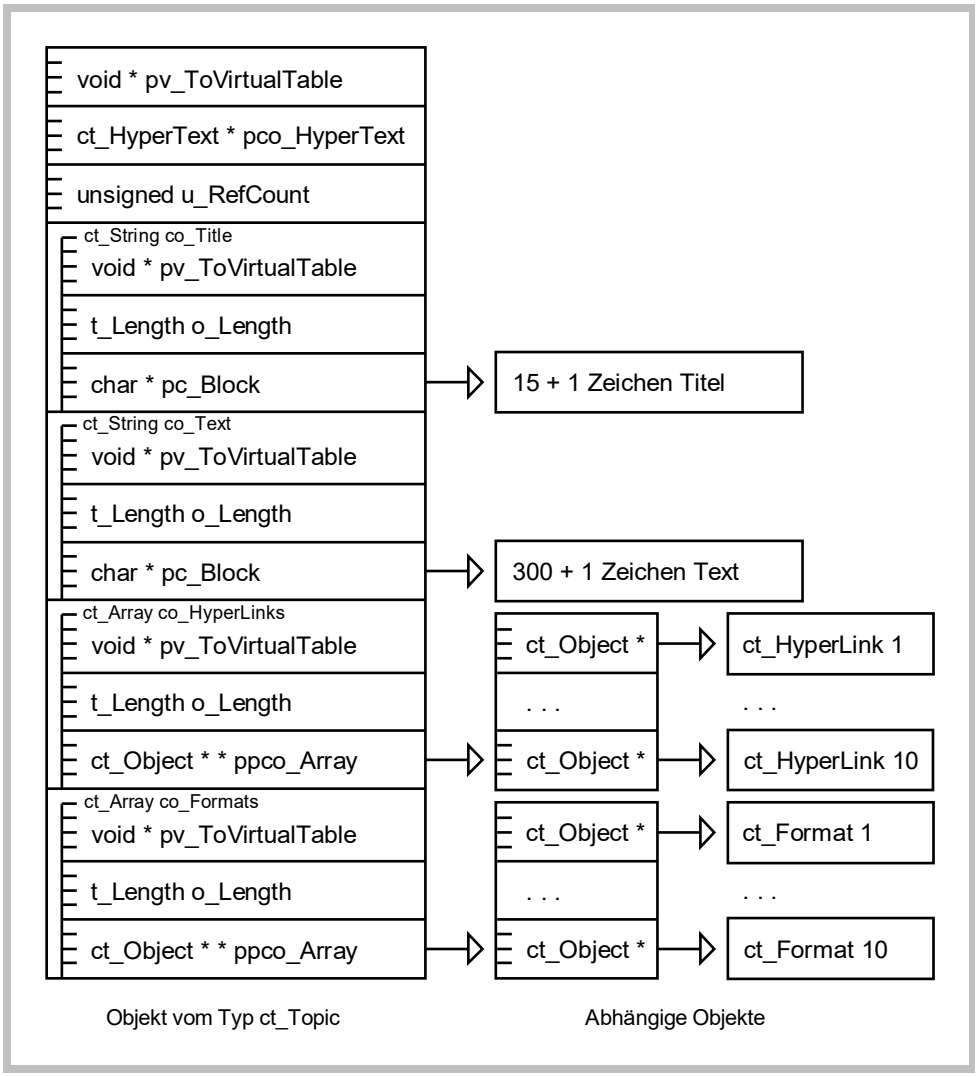

Abb. 1-23: *Speicherlayout einer Instanz der Klasse ct_Topic*

Objekttyp	Absolute Größe	Virt. Tab.-Zeiger	Anzahl Blöcke
Hyperlink	16 Bytes	4 Bytes	1
Formatangabe	16 Bytes	4 Bytes	1
Thema ohne Inhalt	60 Bytes	20 Bytes	1
Titel mit 15 Zeichen	15 + 1 Bytes	0 Bytes	1
Text mit 300 Zeichen	300 + 1 Bytes	0 Bytes	1
10 Array-Einträge	40 Bytes	0 Bytes	1
10 Hyperlinks	160 Bytes	40 Bytes	10
10 Formatangaben	160 Bytes	40 Bytes	10
Thema mit Inhalt	777 Bytes	100 Bytes	25

Tab. 1-3: *Speicheranalyse der Klasse ct_Topic*

Ein Thema mit Inhalt belegt 25 einzelne Speicherblöcke. Die dynamische Speicherverwaltung benötigt pro Block im Durchschnitt acht Bytes für ihre eigenen Zwecke (siehe Abschnitt 1.6.3). Daraus ergibt sich ein tatsächlicher Speicherbedarf von 977 Bytes. Von diesem

Gesamtspeicher entfallen 300 Bytes auf virtuelle Tabellenzeiger und Speicherverwaltung. Das sind *über 30 Prozent* reine Verwaltungsdaten, die für den Programmierer unsichtbar sind. Wegen der großen Anzahl Themen in unserem Datenbeispiel fällt das einzelne Hypertext-Objekt kaum ins Gewicht. Interessant ist aber der Aufwand zur Verwaltung der Themen. Deshalb lassen wir bei der Analyse der Klasse `ct_HyperText` zuerst den Inhalt der Themen außer acht. Danach berechnen wir den Gesamtaufwand mit Inhalt der Themen. Abbildung 1-24 zeigt das Speicherlayout eines Hypertexts. In Tabelle 1-4 befinden sich die Ergebnisse der Speicheranalyse.

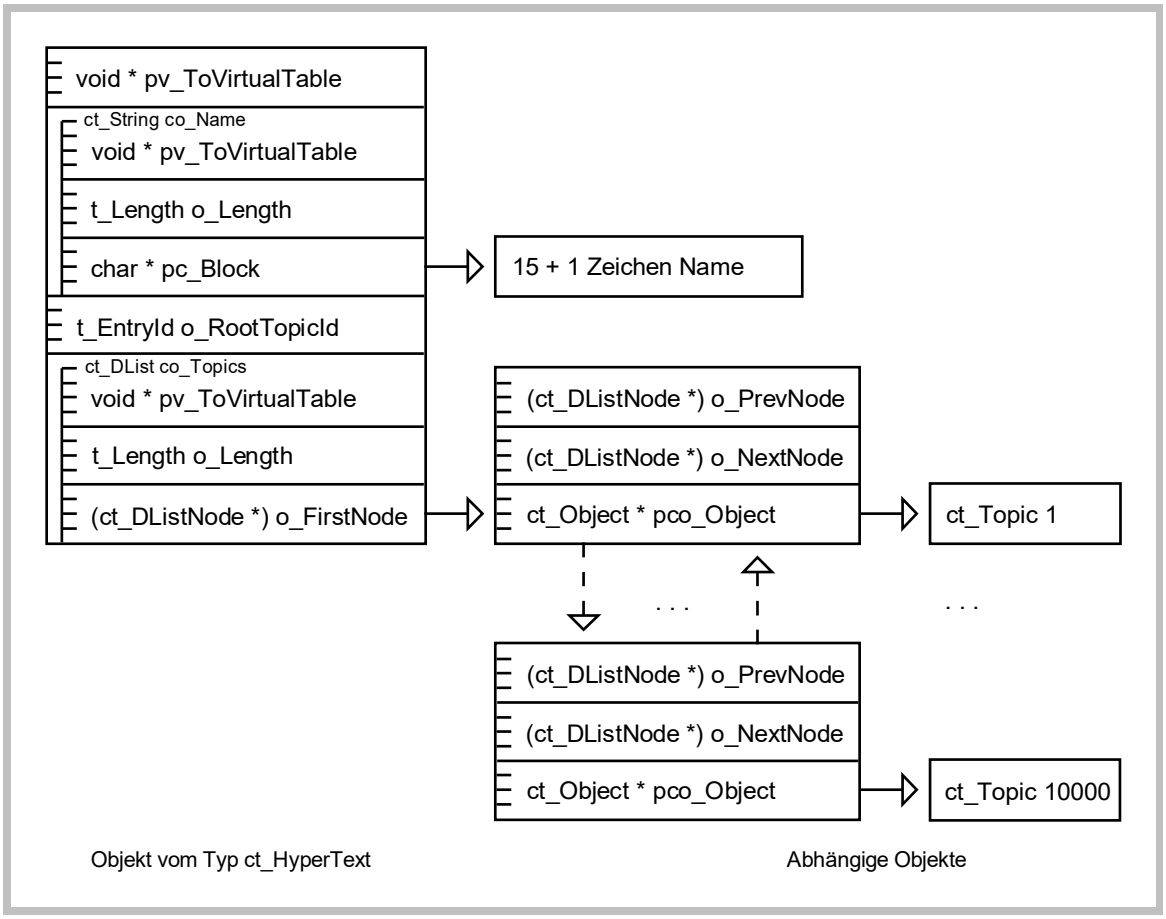

Abb. 1-24: *Speicherlayout einer Instanz der Klasse ct_HyperText*

Objekttyp	Absolute Größe	Virt. Tab.-Zeiger	Anzahl Blöcke
Hypertext ohne Inhalt	32 Bytes	12 Bytes	1
Titel mit 15 Zeichen	15 + 1 Bytes	0 Bytes	1
10 000 DList-Einträge	120 000 Bytes	0 Bytes	10 000
10 000 Themen ohne Inhalt	600 000 Bytes	200 000 Bytes	10 000
Hypertext mit Th. o. I.	720 048 Bytes	200 012 Bytes	20 002
10 000 Themen mit Inhalt	7 770 000 Bytes	1 000 000 Bytes	250 000
Hypertext mit Inhalt	7 890 048 Bytes	1 000 012 Bytes	260 002

Tab. 1-4: *Speicheranalyse der Klasse ct_HyperText*

Bei der Verwaltung der Themen ohne Inhalt fallen die Nachteile einer DList auf. Pro Eintrag sind zwei Speicherblöcke (Node und Objekt) und zwölf Bytes für das Node erforderlich. Die

abschließende Gesamtrechnung zeigt uns, warum beim ersten Test der Speicher übergelaufen ist. Für 260 002 einzelne Speicherblöcke müssen wir etwa zwei MB Zusatzinformationen einkalkulieren. Zusammen mit den virtuellen Tabellenzeigern ergibt sich ein reiner Verwaltungsspeicher von drei MB. Damit haben wir nicht gerechnet. Unsere Hardware ist auf das Entwickeln eines kleinen Beispielprogramms ausgerichtet. Den nächsten Test sollten wir besser auf einer Workstation durchführen.

Auch bei etwas kleineren Datenmengen treten viele einzelne Speicherblöcke auf. Damit ist die beste dynamische Speicherverwaltung überfordert. Wir verstehen nun, warum das Testprogramm im Laufe der Zeit immer langsamer wird. Durch häufiges Ändern der Daten, wie es auch in der Praxis vorkommt, fragmentiert der Speicher zunehmend. Das führt zu einem erhöhten Speicherbedarf und zur Verlangsamung des gesamten Programms.

1.7.3 Auswertung

Die Speicheranalyse hat ergeben, daß unser Programm relativ viel Verwaltungsspeicher verbraucht. Wieviel Speicher wird davon tatsächlich benötigt? Der größte Teil der virtuellen Tabellenzeiger ist in den Hyperlinks und Formatangaben enthalten. Beide Klassen besitzen im Grunde keine virtuellen Methoden. Sie müssen nur wegen unserer Collections von der abstrakten Basisklasse erben. Von der Klasse `ct_Object` erben sie keine Eigenschaften im Sinne einer Is-A-Relation, sondern einigen Overhead in Form des virtuellen Destruktors und der virtuellen Methode `GetTypeName`. Beide Klassen tauchen nie in einem polymorphen Kontext auf und benötigen keinen virtuellen Destruktor.

Die Situation ist mit einem Büro vergleichbar, in dem nur 380-V-Steckdosen vorhanden sind. Für Handwerker mit schweren Maschinen mag es geeignet sein. Was aber tun wir Softwareentwickler, wenn wir in diesem Büro arbeiten müssen? Sollen wir alle elektrischen Kleingeräte mit einem Drehstromanschluß versehen, einschließlich der Kaffeemaschine? Oder ist es nicht besser, das Büro mit einem normalen Stromnetz auszustatten? Die folgende Regel verdeutlicht diesen Sachverhalt.

Erben speicherplatzkritische Klassen von einer Basisklasse mit virtuellen Methoden, prüfen wir, ob die Vererbung inhaltlich erforderlich ist. Erfolgt sie nur aus formalen Gründen, versuchen wir, sie zu umgehen.

Die große Zahl der Speicherblöcke im Testbeispiel wird im wesentlichen durch die Hyperlinks und die Formatangaben verursacht. Auch dieser Verwaltungsaufwand ist im Grunde unnötig. Beide Klassen werden nur in homogenen Collections verwendet. Unsere Collections sind auf diesen Fall nicht eingerichtet. Die verwalteten Objekte werden dynamisch erzeugt. Die dynamische Speicherverwaltung ist jedoch für viele gleichgroße Objekte ungeeignet. Besser wäre es, wenn sich die Collections selbst um die Verwaltung ihrer Objekte kümmern würden.

In unserem Beispielprogramm OHelp kommen keine polymorphen Collections vor. Das entspricht natürlich nicht der Praxis. In großen Anwendungsprogrammen treten sowohl polymorphe als auch homogene Collections auf. Die meisten performancekritischen Collections enthalten jedoch viele gleichartige Objekte. Deshalb richten wir auf die homogenen Collections unsere besondere Aufmerksamkeit. An den bisherigen Collections mußten wir in der Performance-Analyse die folgenden gravierenden Mängel feststellen:

- Virtuelle Methoden verlangsamen das Iterieren und den Zugriff.
- Die verwalteten Objekte müssen von einer abstrakten Basisklasse erben.
- Homogene Collections belasten die Speicherverwaltung unnötig.

Es gibt viele Möglichkeiten, die Performance eines Programms zu verbessern. Man kann die Anwendungsklassen daraufhin prüfen, ob sie unnötige Informationen enthalten. Das kann aber nur vor Ort geschehen. Die verwendeten Algorithmen können verbessert werden, zum Beispiel durch angepaßte Sortierverfahren. Allgemeine Datenstrukturen und Algorithmen sind jedoch seit vielen Jahren gründlich erforscht. Was bleibt, ist die Optimierung des Einsatzes der Programmiersprache und der Standardbibliothek. Bei einer so jungen Sprache wie C++ gibt es auf diesem programmtechnischen Gebiet noch viel zu tun. Die folgende Regel faßt die wesentlichen Resultate unserer Analysen zusammen und sagt damit, was wir in den weiteren Abschnitten untersuchen werden.

Die Hauptfaktoren der programmtechnischen Effizienz sind Speicherverwaltung, Objektverwaltung (Collections) und zeitkritische Methoden.

2 Grundlagen einer besseren Performance

Im ersten Teil des Buchs haben wir gesehen, in welche Performance-Fallen wir bei der Entwicklung eines C++-Programms tappen können. Der zweite Teil enthält das Design zur Verbesserung. Kleine Änderungen an der Oberfläche reichen dazu nicht aus. Wir werden grundlegend neue Konzepte kennenlernen. Zuerst werden wir uns überlegen, was eigentlich ein Computerprogramm ist und aus welchen Komponenten es besteht. Die unmittelbar folgenden Abschnitte sind deshalb sehr abstrakt.

2.1 Ein Abstecher in die Philosophie

2.1.1 Modellierung - Wichtiger Bestandteil menschlicher Tätigkeit

Der Mensch konnte bereits in frühen Stadien der gesellschaftlichen Entwicklung Arbeiten ausführen, zu denen kein Tier in der Lage ist. Besonders auffällig ist das hohe Niveau der Handwerke und Künste in solchen Gesellschaften, in denen auch die Wissenschaften einen hohen Stand erreicht haben. Schon damals setzten die Menschen wissenschaftliche Erkenntnisse in ihrer täglichen Arbeit ein. Die Entwicklung und Anwendung der Wissenschaften wird stark von informationsverarbeitenden Prozessen beeinflußt. Ein hohes Niveau der Aufbewahrung und Weitergabe von Informationen wirkt wie ein Katalysator auf die Forschung. Je besser die gesellschaftlichen und technischen Rahmenbedingungen dafür sind, desto besser können sich wissenschaftliche Erkenntnisse entwickeln und ausbreiten.

Jede Wissenschaft hat einen Gegenstandsbereich. In diesem Bereich stellt sie Untersuchungen an und dokumentiert ihre Ergebnisse. Die frühesten Wissenschaften waren noch sehr mit der Natur und der praktischen Tätigkeit des Menschen verbunden. Die Mathematik ist die erste Wissenschaft, deren Gegenstand kein Bestandteil der Natur ist. Sie befaßt sich mit abstrakten Dingen wie Zahlen, Mengen und Aussagen. Heute gibt es viele abstrakte Wissenschaften. Die Informatik zählt mit ihren Teilbereichen auch dazu. Die Ergebnisse einer Wissenschaft werden in Form von Begriffen, Axiomen und Theorien fixiert. Darin ist der untersuchte Gegenstandsbereich modellhaft abgebildet. Der Wert dieser Erkenntnisse besteht in ihrer universellen Verwendbarkeit. Eine wissenschaftliche Erkenntnis bezieht sich auf einen bestimmten Kontext. Trifft man in der Praxis auf denselben Kontext, kann das vorhandene Wissen angewandt werden. Der Geltungsbereich einer Erkenntnis ist abhängig von ihrem Abstraktionsgrad. Je abstrakter sie ist, desto universeller ist sie einsetzbar.

Ein wenig gebildeter Obstbauer kennt zahlreiche Regeln zum Bilden von Obstmengen, zum Beispiel "Fünf Äpfel plus sieben Äpfel ergibt zwölf Äpfel" und "Fünf Birnen plus sieben Birnen ergibt zwölf Birnen". Diese Regeln sind wiederverwendbar. Jedesmal, wenn fünf Äpfel und sieben Äpfel in einen Korb gelegt werden, gilt die erste Regel. Sie sind aber so konkret, daß sie kaum in eine Wissenschaft aufgenommen werden. Die mathematische Regel "5 + 7 = 12" erspart hingegen die vielen Einzelregeln. Sie ist eine abstrakte Rechenregel, die auf beliebige Gegenstände anwendbar ist. Das Obstbeispiel ist so einfach, daß jedes Kind es versteht. Es zeigt aber exemplarisch, wie kompliziertes Wissen zustandekommt und angewandt wird.

Eine Wissenschaft bildet Modelle von ihrem Gegenstandsbereich. Alle Erkenntnisse werden in modellhafter Form dargestellt. Zum Beispiel wird jeder chemischen Substanz eine Formel zugeordnet. In einem Lehrbuch der Chemie finden wir nützliches Wissen über zahlreiche Substanzen. Sie werden aber nicht in einem Reagenzglas mitgeliefert, sondern als chemische Formel abgebildet. Der Wissenschaftler gewinnt im praktischen Experiment oder durch abstraktes Untersuchen **neues Wissen** (siehe Abbildung 2-1). In unserem Obstbeispiel ist eine Zahl die Verallgemeinerung von Anzahl. Eine Anzahl von fünf Äpfeln ist das Konkrete. Die Zahl Fünf ist das abstrakte Modell. Die zugehörige Wissenschaft ist die Mathematik. Sie untersucht abstrakte Modelle wie ganze Zahlen oder logische Aussagen. Die einfache Rechenregel "5 + 7 = 12" gehört ebenso zum mathematischen Wissen wie die Sätze der Aussagenlogik.

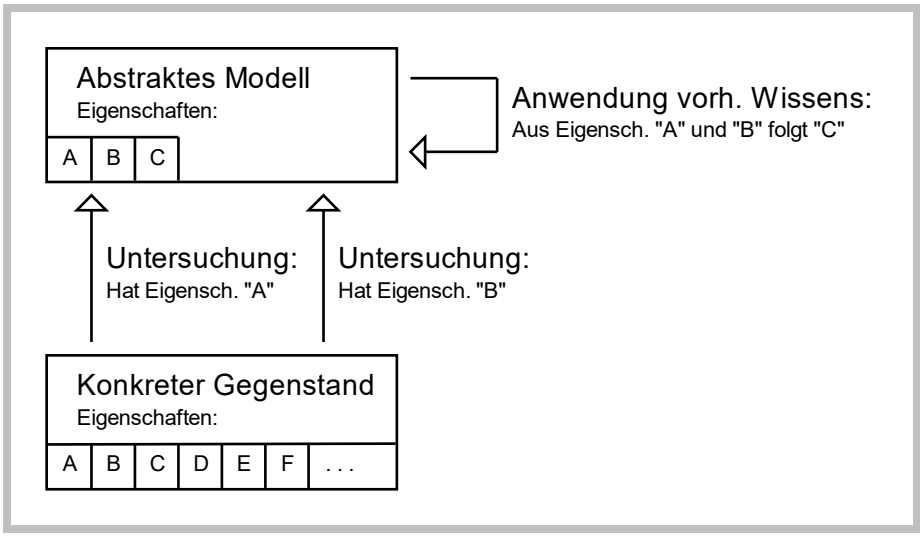

Abb. 2-1: *Bildung neuen Wissens in Form eines Modells*

Auch in der **Anwendung** wissenschaftlicher Erkenntnisse werden Modelle gebildet. Zu einem konkreten Problem suchen wir ein abstraktes Modell. Im günstigen Fall gibt es bereits zahlreiches Wissen über dieses Modell. Dann erhalten wir eine abstrakte Lösung und könnes sie in die Praxis umsetzen (siehe Abbildung 2-2). Haben wir zum Beispiel zwei Chemikalien in Reagenzgläsern vor uns, müssen wir die zugehörigen Formeln bestimmen. Dann können wir nachschlagen, welche Reaktion sich durch Mischen der Substanzen ergibt. Je nach Auskunft des Lehrbuchs beginnen wir mit dem Experiment oder erhöhen den Sicherheitsabstand der Reagenzgläser. Im Obstbeispiel gehört zum konkreten Problem "Fünf Äpfel plus sieben Äpfel" die abstrakte Rechenaufgabe "5 + 7". Dazu finden wir die abstrakte Lösung "12". Die Umsetzung in die Praxis ergibt die konkrete Lösung "Zwölf Äpfel".

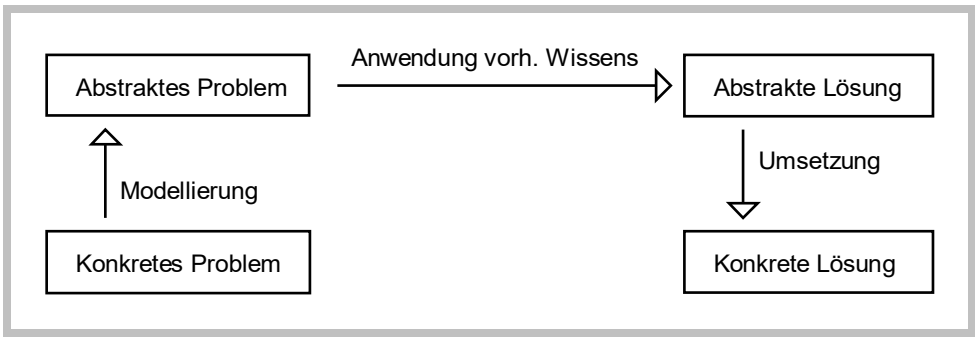

Abb. 2-2: *Anwendung abstrakten Wissens zur Lösung eines Problems*

Wissenschaftliche Erkenntnisse haben zahlreiche Gemeinsamkeiten mit Computerprogrammen. Wir können ein Programm als einen kleinen, in sich geschlossenen Wissensbereich betrachten. Das Wissen wird in einer computerlesbaren Form, zum Beispiel einer Programmiersprache, dargestellt. Ein Programm *entsteht* ähnlich wie neues Wissen (siehe Abbildung 2-1). In der Designphase werden Informationen von konkreten Objekten in ein objektorientiertes Modell übertragen. Vorhandenes "Computerwissen" liegt in Form wiederverwendbarer Bibliotheken vor. Diese werden hauptsächlich während der Implementierung eingesetzt. Auch die *Anwendung* eines Computerprogramms entspricht der Verarbeitung abstrakten Wissens (siehe Abbildung 2-2). Stehen wir vor einer konkreten Aufgabe und verfügen über ein geeignetes Programm, müssen wir das Problem abstrakt formulieren und in den Computer eingeben. Wir erhalten vom Programm eine theoretische Lösung und können sie in die Praxis umsetzen.

2.1.2 Arten und Eigenschaften von Modellen

Im folgenden werden einige Grundbegriffe erläutert. Dabei wird *nicht* versucht, ein neues mathematisches oder philosophisches Begriffssystem aufzustellen. Diese Überlegungen sollen nur unser Verständnis der Computerprogrammierung vertiefen.

Modelle spielen nicht nur bei der wissenschaftlichen Arbeit des Menschen eine Rolle. Im allgemeinsten Sinn ist ein Modell ein Gegenstand, der einem anderen ähnlich ist. Modelle werden gebildet, wenn der Bezugsgegenstand für unser Bewußtsein nicht faßbar ist. Zum Beispiel träumt ein Kind von einer alten Ritterburg. Die Burg ist aber weit entfernt oder nur in einem Märchen vorhanden. Deshalb baut das Kind eine ähnliche Burg in den Sand. Ein Jahr später möchte es ein Flugzeug steuern. Dazu muß das Kind aber noch viele Jahre lernen und begnügt sich mit einem ferngesteuerten Modellflugzeug. Zwischen Sandburg und Flugzeug gibt es einen wesentlichen Unterschied. Das Flugzeug ist ein **dynamisches Modell**, es kann seinen Zustand ändern. Die Burg ändert zwar auch ihren Zustand, wenn ein Haustier darüber läuft. Diese Zustandsänderung ist jedoch keine Modelleigenschaft, sondern eine Eigenschaft des Materials, aus dem das Modell besteht. Als Modell betrachtet besitzt die Sandburg nur statische Eigenschaften.

Modellflugzeug und Sandburg sind reale Gegenstände. Wir können sie sehen und mit den Händen danach greifen. Bei der intellektuellen Tätigkeit nutzen wir jedoch **abstrakte Modelle**. Ein abstrakter Gegenstand besteht nicht aus Materie und kann nur mit Hilfe einer **Darstellung** verwendet werden. Ein sehr einfaches, abstraktes Modell ist die ganze Zahl Vier. Mögliche Darstellungen sind "||||", "IV" und "4". Die Darstellungen sind real, der dargestellte Gegenstand ist jedoch abstrakt. Ein Regelsystem zur Darstellung gleichartiger Gegenstände ist eine **Notation**. Zum Beispiel ist das Dezimalsystem eine Notation für ganze Zahlen. Abstrakte Gegenstände können graphisch oder mit symbolischen Zeichen dargestellt werden. Ein Regelsystem, das auf einer endlichen Zeichenmenge operiert, ist eine **Sprache**.

Wird ein dynamischer Gegenstand abstrakt modelliert, besitzt auch das Modell dynamische Eigenschaften. Das dynamische Verhalten des realen Gegenstands ist ein **Prozeß**. Das entsprechende Modell enthält eine Folge von Zuständen. Die gesamte Folge ist eine **Simulation**. Das Flugzeugmodell ist real und dynamisch. Eine Simulation ist ein abstraktes, dynamisches Modell. Zur Veranschaulichung benötigt sie eine Darstellung. Eine der ersten Anwendungen elektronischer Rechenautomaten war die Berechnung der Flugbahn von Geschossen. Das auf dem Übungsplatz abgefeuerte Geschoß ist ein realer Prozeß. Es bewegt sich nach den Gesetzen den Newtonschen Mechanik und des Luftwiderstands. Diese Gesetze sind lange erforscht und in Lehrbüchern dokumentiert. Durch Programmierung kann die Flugbahn simuliert werden. Der Computer liefert uns Darstellungen der Simulation, zum Beispiel eine Folge von Koordinaten oder eine Zeichnung.

Der Übergang eines abstrakten Gegenstands von einem Zustand in einen anderen wird meist mit **Funktionen** (im mathematischen Sinn eindeutigen Abbildungen) beschrieben. Auch eine

Funktion ist ein abstrakter Gegenstand und benötigt eine Darstellung. Funktionen auf reellen oder komplexen Zahlen können graphisch dargestellt werden. Für exakte Berechnungen nutzt man jedoch Formelschreibweisen. Eine Funktionsdarstellung mit elementaren Einzelschritten ist ein **Algorithmus**. Für das Formulieren von Algorithmen existieren zahlreiche Sprachen. Dazu zählen auch die Programmiersprachen. Abbildung 2-3 zeigt drei Darstellungen derselben Funktion. Eine Funktion ist ein statisches Modell, denn sie ändert nichts, sondern beschreibt etwas. Die Anwendung der Funktion ist jedoch ein dynamisches Modell. Beschreibt die Funktion einen Prozeß, ist ihre Anwendung eine Simulation.

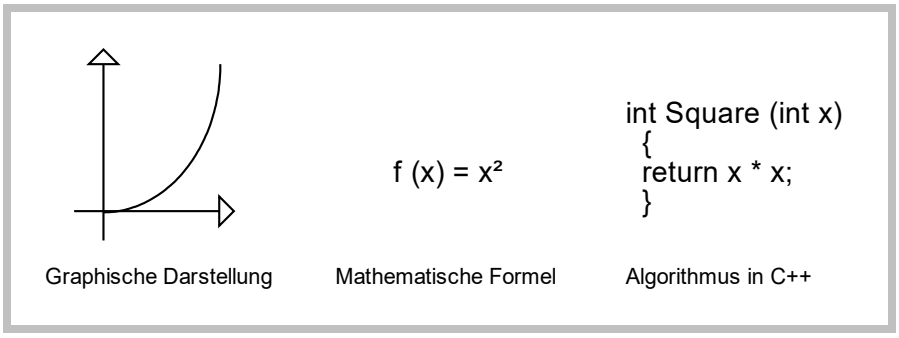

Abb. 2-3: *Verschiedene Darstellungen einer Funktion*

Ein Modell wird zu einem bestimmten Zweck gebildet. Vom Bezugsgegenstand werden nur ausgewählte Eigenschaften auf das Modell übertragen. Bei der Modellierung realer Gegenstände ist eine starke Vereinfachung nötig. Vorgänge in der Natur sind kontinuierlich und besitzen selten scharfe Grenzen. Die zugehörigen Modelleigenschaften sind meist diskret. Sie entsprechen also ganzen Zahlen. Zum Beispiel schaffen Meteorologen statistische Modelle von Wettervorgängen. Zur Auswertung gelangen nicht alle Erscheinungen, die wir in der Atmosphäre beobachten können, sondern nur ausgewählte Meßwerte. Diese werden auf eine sinnvolle Genauigkeit gerundet. Eine rationale Zahl mit einigen Kommastellen kann durch Multiplikation mit einer Zehnerpotenz in eine ganze Zahl umgewandelt werden. Die Wetterstatistiken sind also **diskrete Modelle**.

2.1.3 Modellierung mit Computern

Dieser philosophische Exkurs hat unsere grauen Zellen stark beansprucht. Nun sind wir in der Lage, das Wesen eines **Computerprogramms** zu beschreiben. Es ist ein abstraktes, statisches und diskretes Modell mit Funktionen und Modellen realer Gegenstände. Durch Starten des Programms werden die Funktionen angewendet, und wir erhalten ein dynamisches Modell. Die Darstellung eines Programms erfolgt meist mit einer Programmiersprache. Moderne CASE-Werkzeuge erlauben auch andere textuelle und graphische Ansichten.

Zahlen sind abstrakte Gegenstände und können auf unterschiedliche Weise dargestellt werden. In Computern werden besondere Formen der Darstellung verwendet. Zahlen werden binär codiert, als Folge von Nullen und Einsen. Ein einzelnes Bit entspricht einem physikalischen Zustand in einem Speichermedium. Ein **Computer** ist aus philosophischer Sicht ein technisches Gerät zur Verarbeitung ganzer Zahlen. Er besitzt vier wesentliche Eigenschaften:

- Er kann ganze Zahlen darstellen.
- Auf Zahlen können elementare Operationen angewandt werden.
- Die Operationen sind als Zahlen codiert und können gespeichert werden.
- Gespeicherte Operationen können automatisch ausgeführt werden.

Aus diesen Eigenschaften folgt, daß ein Computer diskrete Modelle darstellen und verändern kann. Das Abarbeiten von Funktionen führt zu einem dynamischen Modell. Wir können also mit einem Computer Prozesse simulieren. Voraussetzung für die Simulation ist, daß die Modelle als ganze Zahlen vorliegen und die Funktionen als elementare Operationen codiert sind. Wer schon einmal einen Computer auf Maschinenebene programmiert hat, möchte kein größeres Programm auf diese Weise entwickeln. Deshalb formulieren wir unsere Modelle in einer höheren Programmiersprache. Mit einem Compiler übersetzen wir das Programm in eine ausführbare Form oder führen es mit einem Interpreter direkt aus.

Bei der theoretischen Lösung eines Problems kümmern wir uns wenig um den erforderlichen Aufwand. Einem Mathematiker reicht es im allgemeinen, wenn ein Problem in endlich vielen Schritten lösbar ist. Soll die Lösung auf einem Computer implementiert werden, müssen wir dessen technische Eigenschaften beachten. Das theoretische Modell muß in einer Programmiersprache dargestellt werden. Moderne, objektorientierte Sprachen erleichtern uns diese Arbeit. Die Ausdrucksmittel entsprechen etwa denen, die wir in unserem abstrakten Denken gebrauchen. Dennoch treffen wir überall auf technische Details. Zum Beispiel werden in modernen Computern ganze Zahlen mit 8, 16 oder 32 Bit dargestellt. Die ganzzahligen Datentypen einer Programmiersprache entsprechen dieser internen Darstellung.

Der Speicher eines Computers besteht aus elementaren Einheiten, den Worten. Ein Wort umfaßt eine bestimmte Anzahl Bits. In ihm kann eine Zahl aus dem entsprechenden Wertebereich gespeichert werden. Jedes Wort hat eine eindeutige **Adresse**. Der Zugriff auf die gespeicherten Zahlen erfolgt stets über ihre Adresse. In älteren Computern war der Adreßraum linear aufgebaut. Die Worte erhielten aufsteigende Nummern. Heute kann der Hauptspeicher mit Auslagerungsdateien vergrößert werden. Auch in einem virtuellen Speicher hat jedes Wort eine eindeutige Adresse.

Ein **Objekt** im Sinne der objektorientierten Programmierung ist ein abstraktes, diskretes Modell. Es enthält binär codierte Informationen und Funktionen, die darauf angewendet werden können. In der objektorientierten Sprechweise sagen wir dazu Attribute und Methoden. Mehrere gleichartige Objekte werden zu einer **Klasse** zusammengefaßt. Die Methoden sind für alle Objekte einer Klasse gleich. Deshalb müssen sie nur einmal gespeichert werden. Die Werte der Attribute können sich jedoch unterscheiden. Dafür benötigt jedes Objekt einen eigenen Speicherbereich.

Wollen wir die Attribute eines Objekts mit einer Programmiersprache beschreiben, müssen wir sie aus primitiven Datentypen zusammensetzen. Für die Darstellung des Objekts im Computer wird eine bestimmte Menge Speicher benötigt. Der Speicherbedarf ist vom Informationsgehalt abhängig. Der **Zugriff** auf das Objekt erfolgt mit der Adresse seines Speichers. Einem einzelnen Objekt können wir einen Namen zuordnen. Die Adresse dieses Objekts benötigen wir nicht. Der Compiler ermittelt sie aus dem Namen. Zur Laufzeit des Programms entstehen jedoch neue Objekte ohne Namen. Wollen wir darauf zugreifen, müssen wir ihre Adresse ermitteln.

Befindet sich ein Objekt in einer Datei, ist der Zugriff komplizierter. Im Hauptspeicher wird eine Kopie erzeugt. Diese besitzt eine Adresse und kann direkt manipuliert werden. Das geänderte Objekt wird in die Datei zurückgeschrieben. Ähnlich wie der Speicher eines Computers besteht auch eine Datei aus Worten. Diese werden vom Dateibeginn aufsteigend numeriert. Innerhalb der Datei hat also jedes Objekt eine eindeutige Adresse. Die Veränderung der Datei erfolgt zwar indirekt, gleicht im Prinzip jedoch der im Hauptspeicher. Zur Vereinfachung betrachten wir deshalb im folgenden keine Dateien und Datenbanken, sondern nur den internen Speicher des Computers.

Während der Abarbeitung eines Programms werden Objekte erzeugt und gelöscht. Dafür benötigen wir eine **Objektverwaltung**. Ihre Aufgabe besteht im Sammeln und Ordnen dynamisch erzeugter Objekte. In diesen Bereich gehören Collections, wie wir sie in den fundamentalen Klassen von OHelp kennengelernt haben. Die Aufbewahrung dieser Informationen erfordert Speicher. Die Objektverwaltung beruht also auf einer

Speicherverwaltung. Diese verarbeitet rohen Speicher und sogt auf unterster Ebene für die Darstellung komplexer dynamischer Modelle in einem Computer.

2.2 Zugriff auf Objekte

Nach diesen allgemeinen Überlegungen betrachten wir nun wieder unsere Programmiersprache. In einem C++-Programm kann ein Objekt auf eine der folgenden Arten entstehen:

- als globale Variable,
- als lokale Variable einer Methode,
- durch den Operator `new`,
- durch einen expliziten Konstruktor-Aufruf,
- als ein temporäres Objekt,
- als Teil eines anderen Objekts (Attribut oder Basisklasse).

In jedem Fall benötigen wir für den Zugriff auf das Objekt seine Adresse. In einem Zeiger oder einer Referenz ist eine Adresse enthalten. Ist das Objekt eine Variable, können wir mit dem Namen darauf zugreifen. Er ist identisch mit einer Referenz. Der Compiler hat für die Variable Speicher bereitgestellt und rechnet den Namen in die Adresse um. Mit dem Adreß-Operator "&" können wir auch die Adresse einer Variablen direkt abfragen. Der Operator `new` liefert einen typisierten Zeiger auf ein neu erzeugtes Objekt. Damit können wir auf das Objekt zugreifen. Durch einen expliziten Konstruktor-Aufruf erhalten wir eine Referenz auf ein temporäres Objekt. Es existiert nur innerhalb der Anweisung, in der es erzeugt wurde, und muß in dieser Anweisung weiterverarbeitet werden. Im Abschnitt 1.4.3 "EntryId und Längenangabe" wurden weitere Beispiele für temporäre Objekte genannt. Wird ein Objekt als Teil eines anderen Objekts erzeugt, erfolgt der Zugriff über die Adresse des umfassenden Objekts. Der folgende Programmausschnitt enthält einige Beispiele für Zugriffe auf Objekte.

```
class ct_Dialog
  {
public:
  void       Show ();
  };

struct st_BigDialog
  {
  ct_Dialog  co_Dialog1;
  ct_Dialog  co_Dialog2;
  };

ct_Dialog TempDialog (); // Methode liefert ein temporäres Objekt

void TestDialog ()
  {
  ct_Dialog co_dialog;           // Lokale Variable
  ct_Dialog * pco_dialog;        // Zeiger auf ein Objekt
  pco_dialog = & co_dialog;      // Berechnung der Adresse der Variablen
  co_dialog. Show ();            // Zugriff mit dem Namen
  ct_Dialog & rco_dialog = co_dialog; // Referenz auf die Variable
  rco_dialog. Show ();           // Zugriff mit der Referenz
  pco_dialog = new ct_Dialog (); // Erzeugen eines neuen Objekts
  pco_dialog-> Show ();          // Zugriff mit dem Zeiger
  (* pco_dialog). Show ();       // Andere Schreibweise für Zeigerzugriff
  ct_Dialog (). Show ();         // Temp. Objekt durch Konstruktor-Aufruf
  TempDialog (). Show ();        // Zugriff auf zweites temporäres Objekt
  st_BigDialog so_bd;            // Umfassende Variable
```

```
so_bd. co_Dialog2. Show ();    // Zugriff auf einen Teil
}
```

2.2.1 Zeiger in C++

Mit einem Zeiger können wir einen Speicherbereich oder ein darin befindliches Objekt identifizieren. Die Schreibweise für einen **Zeigertyp** in C++ ist `declaration_specifier * const_volatile_opt`. Der `declaration_specifier` enthält den Typ, auf den gezeigt wird. Das nachgestellte `const_volatile_opt` bezieht sich auf den Zeiger selbst. Zeigertypen sind zum Beispiel `const char *` (Zeiger auf ein konstantes Zeichen), `int * const` (konstanter Zeiger auf eine Zahl) oder `ct_Dialog *` (Zeiger auf ein Dialogobjekt). Ein Objekt des Zeigertyps ist ein **Zeiger**, zum Beispiel `const char * pc`. Er kann einen beliebigen Wert aus dem Bereich des Zeigertyps annehmen. Ein **Zeigerwert** kann gültig oder ungültig sein. Wir wissen bereits, daß ein C++-Zeiger eine Speicheradresse enthält. Gültige Zeigerwerte sind also Adressen, die von der Speicherverwaltung zur Verfügung gestellt wurden. Der Wert Null ist per Definition ungültig. Die Menge der gültigen Zeigerwerte eines Programms bildet einen **Zeigerraum**.

In C++ muß zwischen typisierten und untypisierten Zeigern unterschieden werden. Ein untypisierter Zeiger enthält im `declaration_specifier` das Schlüsselwort `void`, zum Beispiel `void *` oder `const void * volatile`. Damit können wir kein Objekt, sondern nur einen Speicherbereich identifizieren. Der Compiler wandelt automatisch typisierte in untypisierte Zeiger um. Umgekehrt müssen wir die Typumwandlung selbst vornehmen. Untypisierte Zeiger werden unter anderem beim binären Kopieren von Speicher eingesetzt. Die Standardfunktion `memmove` hat die Deklaration `memmove (void * pv_destination, const void * pv_source, unsigned u_length)`. Wir können sie auch mit typisierten Zeigern aufrufen, zum Beispiel `memmove (& co_dialog1, & co_dialog2, sizeof (ct_Dialog))`.

Die Sprache C++ erlaubt die Verwendung weiterer Zeigertypen. Auf globale Methoden und nichtstatische Member einer Klasse (Attribute und Methoden) können Zeiger gebildet werden. Diese spezialisierten Zeiger betrachten wir nicht näher. Zur Untersuchung der Speicher- und Objektverwaltung sind herkömmliche Zeiger ausreichend. Diese enthalten eine Speicheradresse und haben deshalb zwei wichtige Eigenschaften. Zum einen ist der Zugriff auf die referenzierten Objekte sehr schnell, denn die Hardware des Computers kann Adressen direkt verarbeiten. Zum anderen ist ein Zeigerwert nur zur Laufzeit eines Programms gültig. Beim nächsten Programmstart kann die Speicherverwaltung andere Adressen zur Verfügung stellen. Haben wir ältere Adressen persistent in einer Datei gespeichert, führt deren Verwendung zu Fehlern. Das folgende Programmfragment faßt die wichtigsten Begriffe zusammen.

```
char c;
char * pc;  // Zeigertyp: char *, Zeiger: pc
pc = & c;   // Zeigerwert: Adresse von c
* pc = 'f'; // Zugriff mit dem Zeiger: * pc
```

2.2.2 Indizes von Arrays

Wir können in C++ Arrays von Objekten definieren, zum Beispiel `ct_Dialog aco_dialogs [10]`. Die einzelnen Objekte werden durch einen Index identifiziert. Ein Index kann also wie ein Zeiger betrachtet werden. Der Zeigertyp ist bei jedem Array `unsigned int`. Ein Zeiger ist ein Objekt dieses Typs, zum Beispiel `unsigned int u_idx`. Der Wertebereich des Zeigertyps umfaßt alle nichtnegativen ganzen Zahlen. Im obigen Beispiel sind aber nur die Werte von null bis neun gültig. Die gültigen Zeigerwerte ergeben sich aus der Größe des Arrays und bilden einen Zeigerraum.

Wollen wir auf ein bestimmtes Objekt zugreifen, benötigen wir seine Adresse. Diese erhalten wir durch Indizierung. Zum Beispiel liefert der Ausdruck `aco_dialogs [4]` eine Referenz auf das fünfte Objekt des Arrays. Der Zugriff mit einem Index ist langsamer als mit einem normalen Zeiger. Die Adresse des Objekts muß aus der Anfangsadresse des Arrays und dem Index berechnet werden. Der Index ist jedoch auch bei späteren Programmstarts noch gültig. Er kann also in einer Datei persistent gespeichert und später wieder verwendet werden.

Für den Zugriff auf das fünfte Objekt reicht der Zeigerwert Vier allein nicht aus. Wir müssen auch das Array angeben, auf das er sich bezieht. Das ist ein wichtiger Unterschied zu normalen Zeigern. Während diese *global gültig* sind, ergibt ein Index nur mit dem zugehörigen Array einen Sinn. Das Array übernimmt die Rolle eines Zeigerverwaltungs-Objekts. Im folgenden Programmbeispiel sehen wir die wichtigsten Zeigerbegriffe bei einem Array.

```
ct_Dialog aco_dialogs [10];   // Array mit Zeigerraum 0 bis 9
unsigned int u_idx;           // Zeigertyp: unsigned int, Zeiger: u_idx
u_idx = 4;                    // Zeigerwert: 4
aco_dialogs [u_idx]. Show (); // Zugriff mit Zeiger: aco_dialogs [u_idx]
```

2.2.3 Logische Zeiger

In unserer kleinen Klassenbibliothek haben wir die EntryIds als eine weitere Art Zeiger kennengelernt. Diese besitzen ähnliche Eigenschaften wie die Indizes der C++-Arrays. Der Zeigertyp ist `t_EntryId` und gilt für alle von `ct_Collection` abgeleiteten Klassen. Ein gültiger Zeigerwert ist eine EntryId. Mit der Methode `GetObj` können wir auf das damit referenzierte Objekt zugreifen. Die gültigen EntryIds einer Collection bilden einen Zeigerraum. Mit den Methoden `First` und `Next` können wir ihn vollständig durchlaufen. Eine EntryId ist nur zusammen mit ihrer Collection sinnvoll. Sie besitzt also wie ein Index eine lokale Gültigkeit. Sehen wir uns zur Veranschaulichung wieder ein Programmfragment an.

```
ct_DList co_dlist;         // Liste als Zeigerverwaltungs-Objekt
t_EntryId o_id;            // Zeigertyp: t_EntryId, Zeiger: o_id
o_id = co_dlist. First (); // Zeigerwert: Erster gült. Zeiger der Liste
co_dlist. GetObj (o_id)-> GetTypeName (); // Zugriff mit Methode GetObj
```

Bei der Festlegung des Datentyps für die EntryId haben wir das dynamische Array und die DList berücksichtigt und ihn auf `unsigned long` definiert. Im Performancetest mußten wir feststellen, daß vor allem die Collections Mängel aufweisen. Sie sind für einen möglichst breiten Einsatz konzipiert. Das vereinfacht ihre Anwendung, erschwert jedoch die Anpassung an konkrete Situationen. Die Performance eines Programms wird stark von den fundamentalen Klassen beeinflußt. Sie müssen flexibler als bisher gestaltet werden, damit wir sie an konkrete Erfordernisse, zum Beispiel eine homogene Collection, anpassen können.

In einer DList belegen die Zeiger einen beachtlichen Teil des Speichers. Ein Node enthält je einen Zeiger zum Vorgänger und Nachfolger. Wissen wir von einer Instanz der DList, daß darin nicht mehr als hundert Einträge vorkommen, ist ein 32-Bit-Zeiger ungeeignet. Acht Bit reichen für die Codierung der Zeiger vollkommen aus. Gibt es von dieser Liste viele Instanzen, können wir dadurch viel Speicher sparen. Es ist nicht sinnvoll, einen bestimmten Zeigertyp für alle Collections vorzuschreiben. Normale C++-Zeiger, Indizes und EntryIds werden deshalb zu **logischen Zeigern** verallgemeinert. Diese haben folgende Eigenschaften:

- Ein Zeigertyp ist ein diskreter Datentyp.
- Ein Zeiger kann mit der Zahl Null verglichen werden.
- Es gibt gültige und ungültige Zeigerwerte.
- Der Wert Null ist per Definition ungültig.
- Ein gültiger Zeiger kann in eine Adresse umgerechnet werden.
- Die gültigen Zeigerwerte bilden einen Zeigerraum.
- Dieser wird global oder von einem Objekt verwaltet.

Zeiger können mit beliebigen diskreten Typen dargestellt werden. Sie werden in den fundamentalen und Anwendungs-Klassen eines Programms sehr häufig verwendet. Wir wissen bereits, daß dafür primitive Datentypen effizienter sind als Klassen. Performanceorientierte Zeigertypen sind also `void *`, `unsigned char`, `unsigned short`, `unsigned int` und `unsigned long`.

2.3 Speicherverwaltung

Die Speicherverwaltung hat den Ruf einer betriebssystemnahen Hilfsarbeit. Bei der objektorientierten Umgestaltung der Softwareentwicklung wurde sie bisher außer acht gelassen. Die Situation ist mit einem großen Bürogebäude vergleichbar. Viele betrachten dort den Heizer als einen Hilfsarbeiter im Keller. Nun wird das Haus nach neuen Richtlinien umgestaltet. Rundherum werden neue, größere Fenster eingebaut. Dadurch gelangt mehr Licht in die Räume. Das Arbeiten wird angenehmer. Es wurde aber vergessen, die Heizung an die neuen Bedingungen anzupassen. Besonders bei klirrender Kälte geht durch die Fenster mehr Wärme verloren, und die Leute haben an der Umgestaltung keine Freude mehr.

Das Erzeugen und Löschen von C++-Objekten erfolgt mit den Operatoren `new` und `delete`. Diese sind keine Erneuerung der Speicherverwaltung, sondern nur eine elegante Anwendung. Dynamische Blöcke, zum Beispiel in einer Stringklasse, werden in C++ immer noch mit den Standardfunktionen `malloc`, `realloc` und `free` verwaltet. Unser Performancetest hat gezeigt, daß die Speicherverwaltung eine wichtige Grundlage des Programms ist. Je stärker wir ein Programm algorithmisch optimieren, desto höher werden unsere Ansprüche an die Speicherverwaltung (siehe Abbildung 1-2 im Abschnitt 1.1). Deshalb verleihen wir ihr nun ein objektorientiertes Gewand.

Ein Speicherverwaltungsobjekt nennen wir im folgenden kurz **Store**. Es fordert vom Betriebssystem oder einem anderen Store große Speicherblöcke und verwaltet kleinere. Seine Aufgabe besteht darin, die eigenen Blöcke effizienter zu verwalten als der Lieferant der Blöcke. Das gelingt durch neue Techniken oder durch Spezialisierung. Ist ein Store zum Beispiel auf Blöcke einer bestimmten Größe spezialisiert, kann er sie besser als eine dynamische Speicherverwaltung handhaben. Stores werden wie Collections an performancekritischen Stellen eingesetzt. Die Analyse von OHelp hat ergeben, daß dafür eine gemeinsame Basisklasse mit virtuellen Methoden ungeeignet ist. Von einer Storeklasse fordern wir deshalb nur, daß sie ein Interface wie im folgenden Beispiel besitzt.

```
class ct_AnyStore
  {
public:
  typedef unsigned char  t_Size;
  typedef unsigned short t_Pointer;

  unsigned long        MaxAlloc () const;
  unsigned             StoreInfoSize () const;
  t_Pointer            Alloc (t_Size o_size);
  t_Pointer            Realloc (t_Pointer o_ptr, t_Size o_size);
  void                 Free (t_Pointer o_ptr);
```

```
void *              AddrOf (t_Pointer o_ptr) const;
t_Pointer           LogPtrOf (void * pv_adr) const;
};
```

Der Datentyp `t_Size` wird für die Größe der Speicherblöcke benötigt. Ist er auf `unsigned char` definiert, kann ein Block maximal 255 Bytes umfassen. `t_Pointer` ist der logische Zeigertyp des Stores. Das Beispiel `unsigned short` bedeutet, daß er bis zu 65535 Blöcke verwalten kann. Die ersten beiden Methoden geben über wichtige Eigenschaften des Stores Auskunft. `MaxAlloc` liefert die maximale Größe eines Blocks, zum Beispiel 22 bei einem auf diese Größe spezialisierten Store. Mit `StoreInfoSize` können wir fragen, wieviel Bytes konstanter Verwaltungsspeicher (ohne Rundung) pro Block benötigt werden. Die nächsten drei Methoden gleichen den Standardfunktionen zur Speicherverwaltung. Mit `Alloc` wird ein Block angefordert. Ist die Größe gleich Null oder kann der Block nicht bereitgestellt werden, wird der Nullzeiger zurückgegeben. Die Methode `Realloc` ändert die Größe eines Blocks. Für beide Parameter ist der Wert Null zulässig. Mit `Free` können wir einen Speicherblock freigeben. Auch diese Methode muß den Wert Null als Parameter akzeptieren. `AddrOf` wandelt einen logischen Zeiger in eine Speicheradresse um. Zum logischen Nullzeiger erhalten wir die Adresse Null. `LogPtrOf` berechnet umgekehrt den logischen Zeiger, der zu einer Adresse gehört. Betrachten wir die Anwendung dieser Methoden anhand einfacher Beispiele.

```
ct_AnyStore co_store;          // Store
ct_AnyStore:: t_Pointer o_ptr; // Zugehöriger logischer Zeiger
void * pv;                     // C++-Zeiger, enthält eine Adresse
o_ptr = co_store. Alloc (10);  // 10 Bytes anfordern
pv = co_store. AddrOf (o_ptr); // Adresse des 10-Byte-Blocks ermitteln
o_ptr = co_store. Realloc (o_ptr, 20); // Block vergrößern
pv = co_store. AddrOf (o_ptr); // Neue Adresse ermitteln
co_store. Free (o_ptr);        // Speicher freigeben
pv = 0;                        // Adresse nicht mehr verwenden!
```

Nach jeder Änderung des logischen Zeigers wird die zugehörige Speicheradresse aktualisiert. Wird der Zeiger ungültig, darf die entsprechende Adresse nicht mehr verwendet werden. Eine Speicheradresse kann auch in anderen Fällen ungültig werden. Die Verwaltung der Blöcke in einem Store erfolgt mit Hilfe der logischen Zeiger. Speicheradressen werden nur für den unmittelbaren Zugriff bereitgestellt. Die Gültigkeitsdauer der Adresse ist von der Implementierung des Stores abhängig. Es kann sein, daß das Bereitstellen eines neuen Blocks alle anderen physisch im Speicher verschiebt. Damit ändern sich ihre Adressen. Die logischen Zeiger sind jedoch weiterhin gültig und können für den Zugriff genutzt werden.

2.3.1 Eine runde Sache

Ein großes Problem jeder dynamischen Speicherverwaltung ist die Fragmentierung. Je länger die Liste der freien Blöcke wird, desto langsamer wird die Verwaltung des Freispeichers, und desto mehr Speicher bleibt ungenutzt. Durch Rundung der Blockgröße kann der Fragmentierung entgegengewirkt werden. Beim Blick hinter die Kulissen des Compilers sahen wir, daß schon die Speicherverwaltung der Standardbibliothek auf die nächsthöhere 8- oder16-Byte-Grenze rundet. Werden zum Beispiel 18 Bytes freigegeben und anschließend 22 Bytes angefordert, kann der freigewordene Block genutzt werden. Auch das Verändern der Größe eines Blocks wird durch die Rundung beschleunigt. Verringert sich die Länge einer Zeichenkette von 34 auf 30 Bytes, wird eine 16-Byte-Grenze unterschritten. Ein kleinerer Block muß gesucht und der Inhalt dorthin kopiert werden. Verringert sich hingegen die Länge von 30 auf 26 Bytes, kann derselbe Block genutzt werden. Weder Suchen noch Kopieren sind notwendig.

In fundamentalen Klassen sind oft Rundungsmechanismen eingebaut. Eine Stringklasse kann nicht nur die Länge der Zeichenkette, sondern auch eine Minimalgröße enthalten. Der dynamische Speicherblock wird nie kleiner als die Minimalgröße, auch wenn die eigentliche

Zeichenkette kleiner wird. Dadurch wird die Speicherverwaltung von der Behandlung sehr kleiner Blöcke entlastet. Ist die Minimalgröße ein nichtstatisches Attribut der Stringklasse, kann sie für jede Instanz anders festgelegt werden. Dieser Komfort wird selten benötigt, kostet aber einigen Speicher. Ein Stringobjekt ist klein. Die Hinzunahme eines weiteren Attributs erhöht den relativen Speicherbedarf erheblich. Sinnvoller ist die Deklaration der Minimalgröße als ein statisches Attribut. Die Stringklasse könnte etwa so aussehen.

```
class ct_RoundedString
  {
  static unsigned      u_MinSize;
  unsigned             u_Length;
  char *               pc_Block;

  unsigned             RoundedSize ();
public:
  ....
  };

unsigned ct_RoundedString:: RoundedSize ()
  {
  if (u_Length + 1 < u_MinSize)
    return u_MinSize;
  else
    return u_Length + 1;
  }
```

Das statische Attribut `u_MinSize` gilt für alle Instanzen dieser Stringklasse. Die private Methode `RoundedSize` berechnet eine gerundete Größe. Sie verwendet die Länge, die Minimalgröße und eventuell andere Rundungsverfahren. Bisher wurde ein neuer Block durch die Anweisung `realloc (pc_Block, u_Length + 1)` angefordert. Nun schreiben wir `realloc (pc_Block, RoundedSize ())`.

Speicherblöcke variabler Länge werden nicht nur in Strings, sondern auch in Bitmaps, Arraycollections usw. verwendet. Wollten wir die dynamische Speicherverwaltung durch Rundung effektiv entlasten, müßten wir in vielen Klassen Rundungsmechanismen vorsehen. Die Rundung ist jedoch keine Eigenschaft der dargestellten Modelle. Mit einer Stringklasse wird eine Folge von Zeichen modelliert. Diese besitzt eine bestimmte Länge, aber keine gerundete Größe. Auch das Modell der Zeichenfolge benötigt keine Rundung. Sie ist nur zur Entlastung der Speicherverwaltung erforderlich und gehört somit in deren Arbeitsbereich.

Eine ähnliche Arbeitsteilung wird vom *Model-View-Controller*-Konzept vorgeschlagen. Es beschreibt die Architektur interaktiver Programme. Danach ist es zum Beispiel unzulässig, daß das *Model* eine Fehlermeldung auf dem Bildschirm anzeigt. Die Anzeige der Nachricht und die Auswertung der Benutzereingabe gehören in den Bereich *Controller*. Analog dazu sollten in Strings und Arrays keine Rundungsmechanismen enthalten sein. Diese Klassen benötigen Speicher zur Darstellung ihrer Informationen. Das effektive Verwalten des Speichers ist jedoch nicht ihre Aufgabe. Die Rundung der Größe dynamischer Blöcke steht auf derselben Ebene wie deren Anfordern und Freigeben. Es gehört in den Bereich der Speicherverwaltung. Ändern wir nach diesem Konzept die Stringklasse.

```
class ct_RoundedStore
  {
public:
  typedef unsigned int t_Size;
  typedef void *       t_Pointer;
private:
  t_Size               o_MinSize;
  t_Size               Round (t_Size o_size);
public:
  ....
  } co_RoundedStore;
```

```
class ct_RoundedString
  {
  ct_RoundedStore:: t_Size    o_Length;
  ct_RoundedStore:: t_Pointer o_Block;
public:
  ....
  };

const char * ct_RoundedString:: GetStr ()
  {
  return (const char *) co_RoundedStore. AddrOf (o_Block);
  }

void ct_RoundedString:: Insert (unsigned u_pos, const char * pc_ins)
  {
  ....
  o_Block = co_RoundedStore. Realloc (o_Block, o_Length + 1);
  ....
  }
```

In diesem Beispiel wird die spezialisierte Storeklasse `ct_RoundedStore` deklariert. Sie enthält das Attribut `o_MinSize`. Die private Methode `Round` rundet eine gegebene Größe mit Hilfe der Minimalgröße und eventuell anderer Verfahren. Sie wird von den Methoden `Alloc` und `Realloc` aufgerufen. Die Klasse `ct_RoundedStore` kann als globale oder lokale Variable oder als Teil anderer Klassen verwendet werden. Das Objekt `co_RoundedStore` realisiert eine globale Verwaltung der Minimalgröße und der gerundeten Blöcke. Die Stringklasse `ct_RoundedString` zeigt eine Anwendung des Stores. Die Attribute `o_Length` und `o_Block` richten sich nach den Typen `t_Size` und `t_Pointer` der Storeklasse. Zum Verändern der Größe des Blocks wird die Methode `ct_RoundedStore:: Realloc` aufgerufen. Für dynamische Arrays kann dasselbe oder ein anderes globales Storeobjekt verwendet werden.

2.3.2 Rundungstechniken

Die Rundung spart auf der einen Seite Speicher, denn die Liste der freien Blöcke wird kleiner. Auf der anderen Seite kostet sie Speicher, denn die belegten Blöcke sind größer als nötig. Zum Beispiel bleiben bei der Rundung von 67 auf 80 Bytes die letzten 13 Bytes ungenutzt. Der **Gesamtspeicherbedarf** eines Programms ergibt sich aus der Summe der belegten und freien Blöcke. Ist er gleich oder geringfügig höher als bei der ungerundeten Speicherverwaltung, haben wir dennoch einen Performancegewinn erzielt. Die Rundungstechniken sorgen für eine kleinere Freiliste. Dadurch wird die Verwaltung des freien Speichers wesentlich beschleunigt. Abbildung 2-4 verdeutlicht diese Verhältnisse.

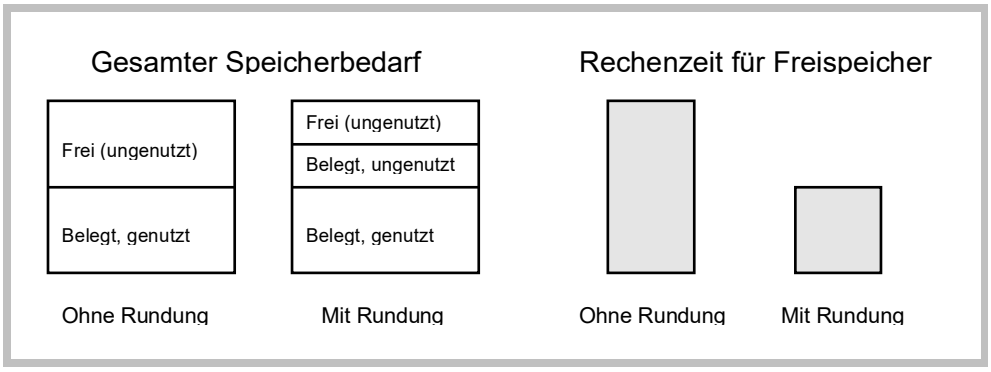

Abb. 2-4: *Auswirkung der Rundung auf die Performance*

Welche Rundungstechnik für ein Anwendungsprogramm die beste ist, hängt von dessen Speicheranforderungen ab. Belegt es zunehmend neuen Speicher und gibt wenig Speicher frei, sollte keine Rundung eingesetzt werden. Die Freiliste ist klein und kann wenig optimiert werden. Die Rundung würde aber die belegten Blöcke vergrößern. Je mehr Speicher das Programm freigibt und neu anfordert, desto besser wirkt sich die Rundung auf die Performance aus. Eine grobere Rundung erhöht den Anteil des belegten, ungenutzten Speichers, verringert aber die Fragmentierung. Je stärker die Belastung der Speicherverwaltung ist, desto grober sollte die Rundungstechnik sein (siehe Abbildung 2-5 und Rechenzeittest weiter unten).

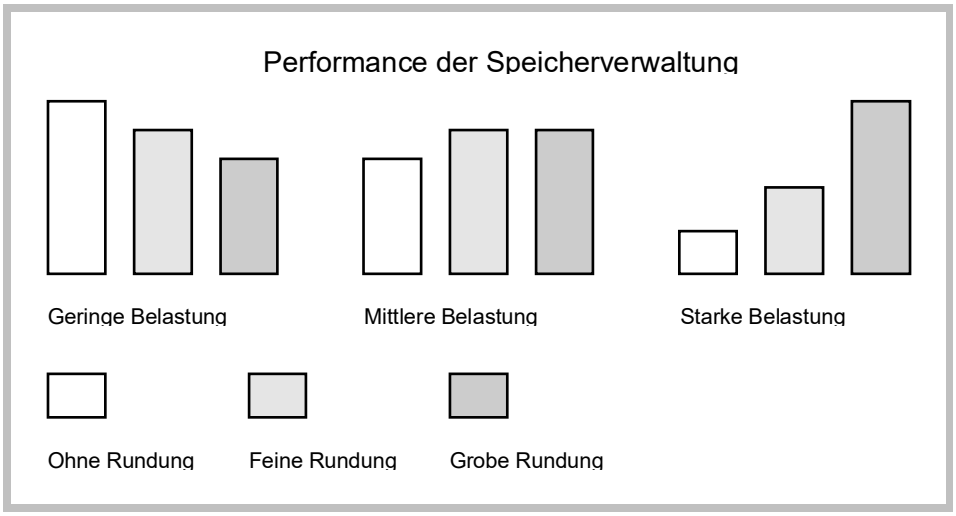

Abb. 2-5: *Rundungstechniken bei zunehmender Belastung*

Ziel einer Rundungstechnik ist es, die Anzahl der Blockgrößen zu minimieren. Je weniger Blockgrößen es gibt, desto höher ist die Wahrscheinlichkeit, in der Freiliste ein passendes Element zu finden. Gibt es für eine Anforderung keinen freien Block, muß neuer Speicher verwendet werden. Ohne Rundung existieren zwischen ein und hundert Bytes genau hundert mögliche Blockgrößen. Werden 67 Bytes angefordert, ist die Wahrscheinlichkeit gering, daß die Freiliste einen Block dieser Größe enthält. Die Verwendung einer **Schrittweite** ist eine einfache Rundungstechnik. Werden die Blöcke zum Beispiel auf eine 16-Byte-Grenze gerundet, gibt es zwischen ein und 128 Bytes nur noch acht verschiedene Größen (16, 32, 48, 64, 80, 96, 112 und 128 Bytes). Die Anforderung 67 wird auf 80 Bytes gerundet. Die Wahrscheinlichkeit, in der Freiliste einen 80iger Block zu finden, ist größer.

Wir haben die Festlegung einer **Minimalgröße** bereits als eine Rundungstechnik kennengelernt. Ergänzen wir die Schrittweite 16 um die Minimalgröße 50 Bytes, existieren zwischen ein und 128 Bytes nur noch die Größen 64, 80, 96, 112 und 128 Bytes. Damit wird die Speicherfragmentierung weiter eingeschränkt. Der Gesamtspeicherbedarf erhöht sich jedoch. Alle Anforderungen von ein bis 64 Bytes werden auf 64 Bytes gerundet. Besonders bei kleinen Blöcken ist der relative Anteil ungenutzten Speichers hoch. Eine Minimalgröße lohnt also nur, wenn die Anzahl der kleinen Blöcke gering ist.

Die Rundung auf eine 16-Byte-Grenze ist im Bereich von ein bis 128 Bytes sinnvoll. Bei größeren Blöcken wirkt sie der Fragmentierung kaum noch entgegen. Zum Beispiel nützt es wenig, einen Block von 375 auf 384 Bytes zu runden. Eine gute Rundungstechnik sollte sich der Größe der Blöcke anpassen. Wir benötigen eine **relative Schrittweite**. Zwischen 128 und 256 Bytes kann die Schrittweite auf 32 erhöht werden, zwischen 256 und 512 Bytes auf 64 usw. Die Berechnung der gerundeten Größe wird dadurch komplizierter. Zu einem gegebenen Wert (375) müssen wir die nächstkleinere Zweierpotenz ermitteln (256). Diese wird durch den **Schritt-Teiler** Vier dividiert, und wir erhalten die Schrittweite (64). Wir könnten uns auch nach Zehnerpotenzen richten. Die Berechnung von Zweierpotenzen ist

jedoch schneller, denn sie läßt sich auf einfache Shift-Operationen zurückführen. In Abbildung 2-6 sehen wir relative Schrittweiten bei verschieden großen Schritt-Teilern.

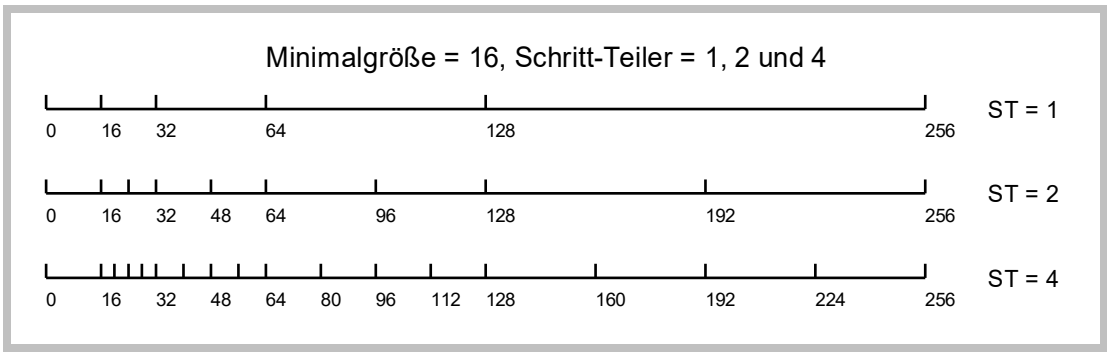

Abb. 2-6: *Auswirkung des Schritt-Teilers auf die Rundung*

Zur Vereinfachung der Berechnung fordern wir, daß sowohl Minimalgröße als auch Schritt-Teiler Zweierpotenzen sind. Ein Schritt-Teiler, der größer als die Minimalgröße ist (zum Beispiel 32 und16), ergibt keinen Sinn. Er ist in unserem Rundungsverfahren nicht erlaubt. In Abbildung 2-6 haben wir für die Berechnung der Schrittweite einen sturen Algorithmus eingesetzt. Er berechnet beim Schritt-Teiler Vier zwischen 16, 32 und 64 die Schrittweiten Vier und Acht. Auch dieses Verhalten ergibt keinen Sinn. Wir ändern den Algorithmus, so daß die Schrittweite nicht kleiner als die Minimalgröße werden kann (siehe Abbildung 2-7).

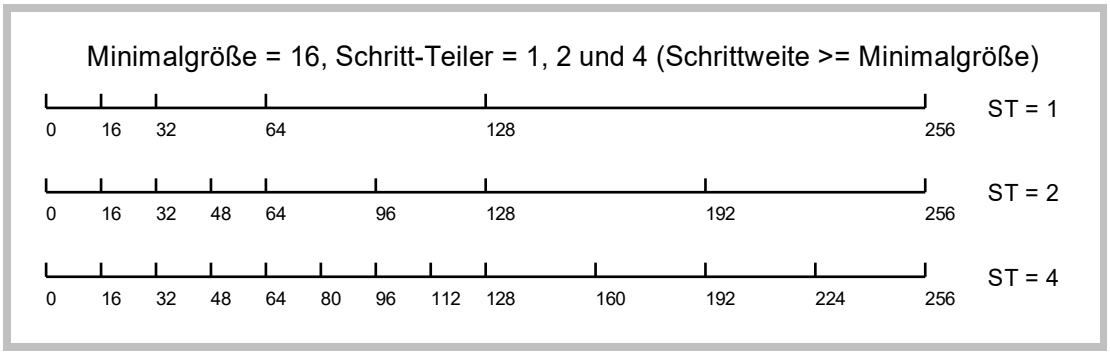

Abb. 2-7: *Verbesserts Rundungsverfahren*

Ein größerer Schritt-Teiler (8, 16 usw.) verkleinert die Schrittweite und erhöht die Anzahl möglicher Blockgrößen. Damit passen sich die Blöcke den tatsächlichen Anforderungen besser an, die Fragmentierung nimmt aber wieder zu. Beim kleinstmöglichen Schritt-Teiler Eins ergibt sich hingegen die **binäre Rundung**. Dabei werden alle Anforderungen auf die nächstgrößere Zweierpotenz gerundet. Der Anteil des pro Block ungenutzten Speichers ist größer als bei anderen Rundungsverfahren. Die binäre Rundung wirkt aber der Fragmentierung am stärksten entgegen. Sie ist besonders für Programme mit vielen Anforderungen und Freigaben geeignet.

An einem rechen- und speicherintensiven Programm wurde der folgende Test durchgeführt: Es wurde einmal ohne und einmal mit binärer Rundung gestartet. Am Programm selbst wurde nichts geändert. Es wurde nur die Speicherverwaltung der C-Standardbibliothek durch einen Store mit binärer Rundung ersetzt. Der Gesamtspeicherbedarf blieb etwa gleich. Beim zweiten Mal lief das Programm aber 2,5 mal schneller. Das bedeutet, daß beim ersten Programmstart etwa 70 Prozent der Rechenzeit auf die Standardfunktionen `malloc`, `realloc` und `free` entfielen (siehe Abbildung 2-8).

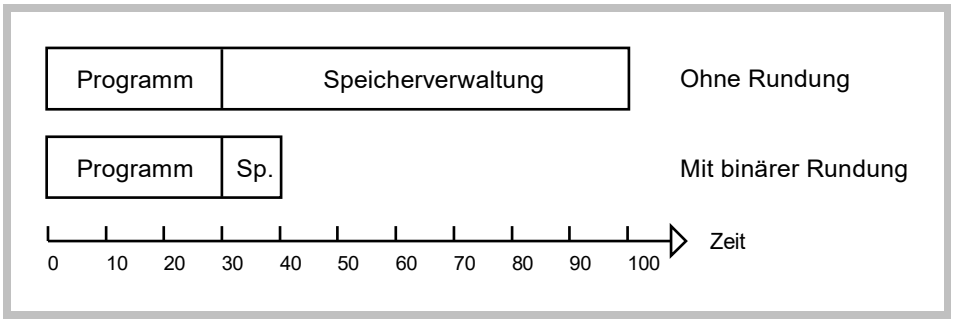

Abb. 2-8: *Anteile der Gesamtrechenzeit*

Neben den genannten Rundungstechniken sind weitere denkbar. Eine Kombination aus Minimalgröße, relativer Schrittweite und variablem Schritt-Teiler ist jedoch für die meisten Fälle ausreichend. Vor allem mit dem Schritt-Teiler können wir die Rundung an konkrete Erfordernisse anpassen. Wird die Speicherverwaltung wenig beansprucht, wählen wir einen großen Schritt-Teiler (z.B. 16). Bei starker Belastung verwenden wir mit dem Schritt-Teiler Eins die binäre Rundung. In der Praxis müssen wir beachten, daß jede Rundungstechnik gleichmäßig verteilte Speicheranforderungen voraussetzt. Eine konkrete Anwendung ist darauf zu prüfen, ob sie viele ungünstige Blöcke anfordert. Zum Beispiel sind bei der Schrittweite 16 die Werte 17, 33, 49 usw. ungünstig.

Jede dynamische Speicherverwaltung beansprucht pro Block einige Bytes Verwaltungsspeicher. Wir können ihn von einem Storeobjekt mit der Methode `StoreInfoSize` abfragen. Bei der Berechnung der gerundeten Größe muß er berücksichtigt werden. Erst dann wirkt die Rundung effektiv der Fragmentierung entgegen. Im folgenden Rechenbeispiel beträgt die Schrittweite 16 und die `StoreInfoSize` vier Bytes. 20 Bytes sollen freigegeben und danach zweimal zehn Bytes angefordert werden. Ohne Berücksichtigung der `StoreInfoSize` werden 20 Bytes auf 32 gerundet. Die interne Größe des Blocks beträgt 36 Bytes. Zehn Bytes werden auf 16 gerundet. Benötigt werden also zwei Blöcke der internen Größe 20, insgesamt 40 Bytes. Dafür reicht der freigegebene Speicher von 36 Bytes nicht aus, und neuer Speicher muß verwendet werden. Berücksichtigen wir die `StoreInfoSize`, werden 20 Bytes auf 28 gerundet. De facto werden 32 Bytes freigegeben. Zehn Bytes werden nun auf zwölf gerundet und zwei neue Blöcke der internen Größe 16 benötigt. Dafür kann der freigewordene Block genutzt werden (siehe Abbildung 2-9).

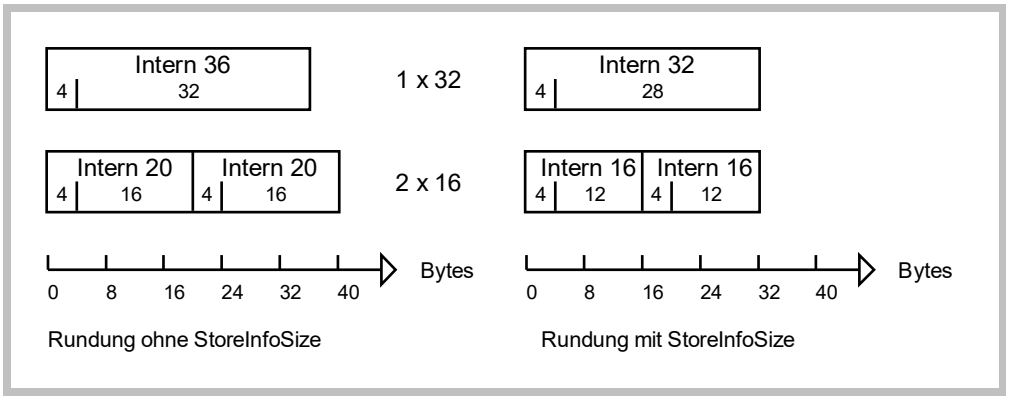

Abb. 2-9: *Rundung und StoreInfoSize*

2.3.3 Feste Speicherverwaltung

Eine dynamische Speicherverwaltung verursacht pro Block einen doppelten Overhead. Die angeforderte Größe wird gerundet, und einige Bytes Verwaltungsspeicher werden

hinzugefügt. Bei Anforderung vieler Blöcke derselben Größe ist eine feste Speicherverwaltung günstiger. Sie ist auf Blöcke einer bestimmten Größe spezialisiert. Im einfachsten Fall werden sie direkt hintereinander in einem größeren, dynamischen Block untergebracht. Betrachten wir auch dazu ein Rechenbeispiel. Die Blockgröße betrage zehn Bytes. Die dynamische Speicherverwaltung benötigt dafür 16 Bytes, für fünf Blöcke insgesamt 80 Bytes. Werden die fünf mal zehn Bytes in einem einzelnen Block hintereinander gespeichert, entsteht ein Block der internen Größe 64 Bytes, also 16 Bytes weniger (siehe Abbildung 2-10).

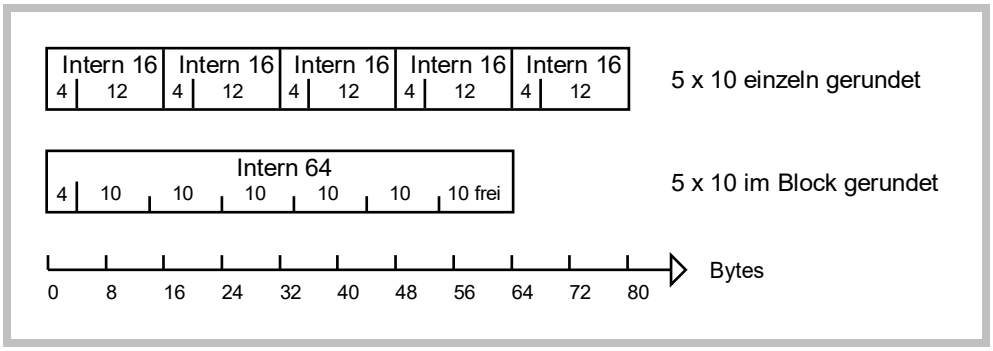

Abb. 2-10: *Dynamische und feste Speicherverwaltung*

Im Durchschnitt ist die feste Speicherverwaltung effektiver als die dynamische. Es gibt jedoch auch Grenzfälle. Beträgt zum Beispiel die Blockgröße zwölf Bytes, entsteht bei der dynamischen Verwaltung kein Rundungs-Overhead. Für fünf Blöcke belegt sie 80 Bytes, die feste Verwaltung 64 Bytes. Kommt ein Block hinzu, benötigt die dynamische 96 Bytes. Die feste belegt nun in einem Block 72 Bytes. Bei binärer Rundung wird er intern auf 128 Bytes gerundet. Für acht Blöcke benötigen beide denselben Speicher (128 Bytes). Ab neun Blöcken ist die feste Speicherverwaltung wieder günstiger. Für einen Block der Größe 14 Bytes benötigt die dynamische Verwaltung intern 32 Bytes. Der Overhead beträgt pro Block 18 Bytes. Bei dieser Blockgröße ist die feste Verwaltung deutlich sparsamer (siehe Abbildung 2-11), ebenso bei sehr kleinen Blöcken von vier oder acht Bytes. Dynamische Speicherverwaltungen besitzen meist eine Minimalgröße von 16 Bytes und sind auf kleinere Blöcke nicht eingerichtet.

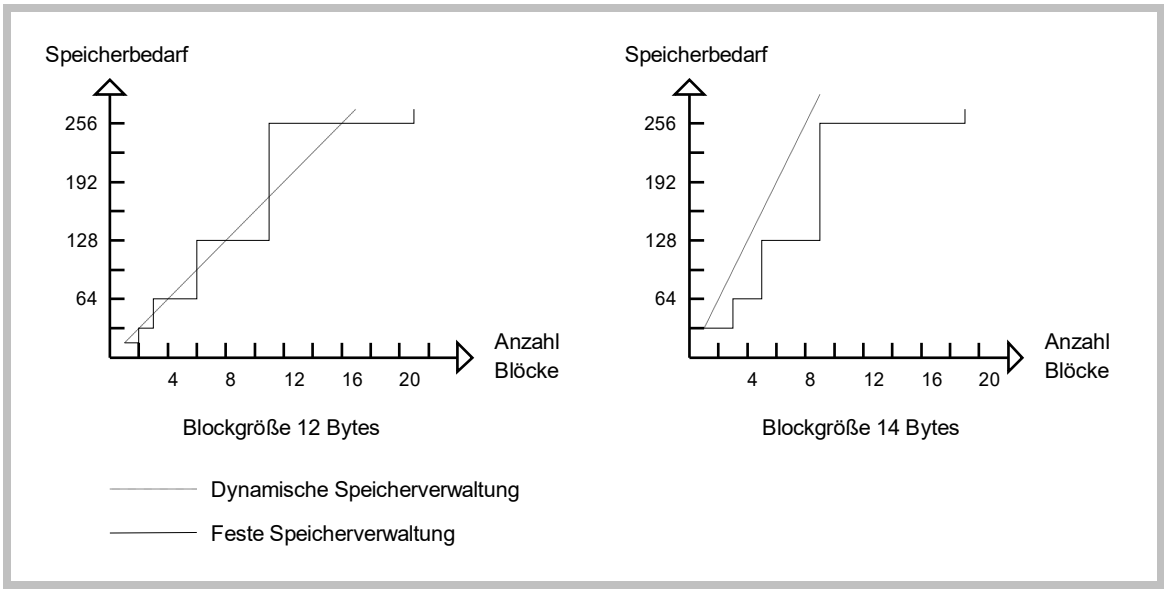

Abb. 2-11: *Speicherbedarf bei 12- und 14-Byte-Blöcken*

Eine Speicherverwaltung darf ihre Blöcke physisch verschieben. Dabei ändern sich deren Adressen. Die logischen Zeiger müssen jedoch ihre Gültigkeit behalten. Die einfachste Zuordnung logischer Zeiger zu den Blöcken einer festen Speicherverwaltung ist die Indizierung. Die fortlaufende Nummer innerhalb des umfassenden Blocks ist zugleich der logische Zeiger. Die Zählung muß mit der Nummer Eins beginnen, denn der Zeigerwert Null ist per Definition ungültig. Die Verwaltung dieser Zeiger erfordert keinen Zusatzaufwand. Die Blöcke dürfen aber nicht einzeln verschoben werden. Dadurch würde sich ihr Index ändern. Auch bei einer festen Speicherverwaltung können beliebige Blöcke freigegeben werden. Befindet sich der freizugebende Block am Ende des umfassenden Blocks, kann dieser verkleinert werden. Für andere Blöcke wird eine Freiliste angelegt, denn die nachfolgenden Blöcke müssen ihren Index behalten (siehe Abbildung 2-12).

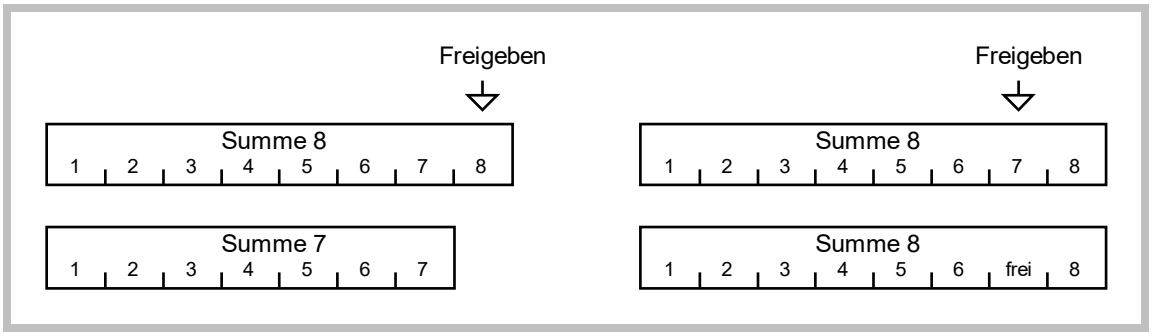

Abb. 2-12: *Freigabe bei fester Speicherverwaltung*

Die freien Blöcke haben alle dieselbe Größe. Deshalb ist die Behandlung der Freiliste nicht schwer. Ein freier Block benötigt keine Größenangabe, sondern nur einen Verweis auf das nächste Element der Freiliste. Es entstehen jedoch einige Nachteile im Vergleich zur dynamischen Verwaltung. Für eine neue Anforderung sollte der freie Block mit dem kleinsten Index verwendet werden. Dadurch erhöht sich die Wahrscheinlichkeit, daß am Ende des umfassenden Blocks etwas frei wird und dieser verkleinert werden kann. Die Freiliste muß also sortiert sein. Bei einer dynamischen Speicherverwaltung werden benachbarte Freiblöcke zusammengefaßt und können für größere Anforderungen genutzt werden. Die freien Blöcke einer festen Verwaltung stehen nur für Anforderungen derselben Größe zur Verfügung. Abbildung 2-13 zeigt ein Beispiel für eine sortierte Freiliste.

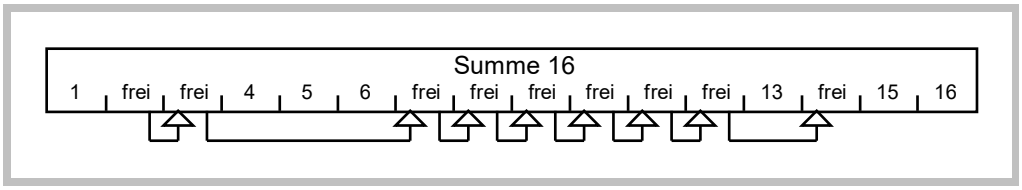

Abb. 2-13: *Freiliste im umfassenden Block*

Der umfassende Block kann nur verkleinert werden, wenn der physisch letzte Block freigegeben wird. Treten freie Blöcke in der Mitte auf, bleibt der Speicher ungenutzt. Je mehr Blöcke freigegeben werden, desto größer wird dieses Problem. Eine Lösung dafür ist die Aufteilung des Festspeichers auf mehrere Blöcke. Diese können mit einer Arraycollection verwaltet werden. Pro Teilblock entsteht der Overhead eines zusätzlichen Zeigers. Sind die Teilblöcke größer als hundert Bytes, fällt er kaum noch ins Gewicht. Die Verwaltung der Freiliste wird jedoch wesentlich effizienter. Praktische Tests haben ergeben, daß 250 bis 1000 Elemente pro umfassendem Block optimal sind. Treten mehr auf, sollte er in Teilblöcke zerlegt werden (siehe Abbildung 2-14).

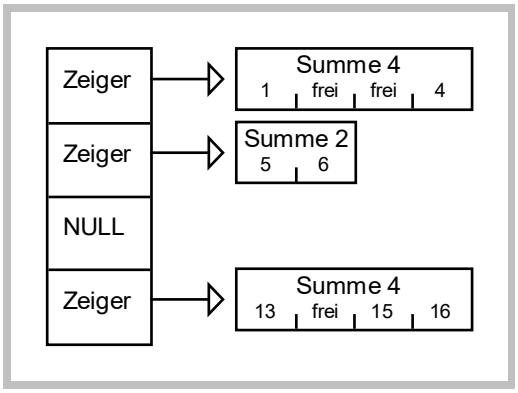

Abb. 2-14: *Aufteilung auf mehrere Blöcke*

2.4 Objektverwaltung

In der objektorientierten Sprechweise ist ein Programm eine Sammlung von Objekten. Mit einem Computer können wir Objekte speichern und deren Methoden aufrufen. Dabei entstehen neue Objekte, und alte werden gelöscht. Einige Objekte sind bereits im statischen Modell eines C++-Programms enthalten. Dazu zählen Variable und temporäre Objekte. Die Bereitstellung von Speicher und die Verwaltung ihrer Adressen übernimmt der Compiler. Globalen Variablen und statischen Attributen der Klassen werden beim Programmstart feste Speicherbereiche zugewiesen. Lokale Variable und temporäre Objekte legt der Compiler auf dem Stack ab. Um dynamisch erzeugte Objekte müssen wir uns selbst kümmern. Die Speicherverwaltung stellt den erforderlichen Speicher zur Verfügung. Für das Sammeln und Ordnen ist die **Objektverwaltung** zuständig.

Ihre theoretische Grundlage ist die Mengenlehre. Nach der Art der Elemente unterscheiden wir homogene und polymorphe Mengen. Eine homogene Menge enthält nur gleichartige Objekte, in einer polymorphen können Objekte verschiedener Typen zusammengefaßt werden. Sind die Elemente einer Menge einzeln bekannt, sprechen wir von einer statischen Menge. Sie kann im statischen Modell eines C++-Programms mit einem Datentyp beschrieben werden. Ein C++-Array ist zum Beipiel eine statische homogene Menge. Anzahl und Typ der Elemente sind bekannt. Sie werden mit einem Index identifiziert. Statische polymorphe Mengen können mit Strukturen (`struct`) und Klassen (`class`) dargestellt werden. Darin besitzt jedes Element (Attribut) einen eindeutigen Namen.

Das Gegenteil einer statischen ist eine dynamische Menge. Anzahl und Typ ihrer Elemente ergeben sich erst zur Laufzeit des Programms. Die Beschreibung dynamischer Mengen ist kein Bestandteil der Sprache C++. Da sie jedoch häufig verwendet werden, existieren zahlreiche Implementierungen in Klassenbibliotheken. In den fundamentalen Klassen unseres Beispielprogramms OHelp haben wir zwei Arten dynamischer Mengen kennengelernt, die Collections `ct_Array` und `ct_DList`. Damit können wir polymorphe Mengen modellieren. Der Performancetest hat gezeigt, daß dieses Collection-Konzept nicht ausreicht. In einem Programm treten auch viele dynamische homogene Mengen auf. Zur Unterscheidung von den Collections nennen wir sie im folgenden **Container** (siehe Tabelle 2-1).

	Homogene Menge	Polymorphe Menge
Statische Menge	C++-Array	Struktur und Klasse
Dynamische Menge	Container	Collection

Tab. 2-1: *Modellierung von Mengen in C++*

Ein Container ist auf Objekte eines bestimmten Typs spezialisiert. Wir können ihn als Low-Level-Collection bezeichnen, denn er ist nicht so universell wie eine Collection einsetzbar. Diese bietet mehr Programmierkomfort. In den Bereich der Objektverwaltung gehören auch Datenbanken. Sie besitzen einen noch größeren Funktionsumfang. Datenbanken dienen der persistenten Verwaltung von Objekten in einem Netzwerk. Eine Behandlung dieses Themas geht aber weit über das Ziel des Buchs hinaus. Wir konzentrieren uns auf Dinge, die sich mit den sprachlichen Mitteln von C++ darstellen lassen. Dazu gehören Container und Collections. Sie bilden den Kern der Verwaltung dynamischer Objekte und haben einen großen Einfluß auf die Performance. Abbildung 2-15 zeigt den Aufbau der Objektverwaltung und ihre Position innerhalb der Verwaltungshierarchie eines Programms.

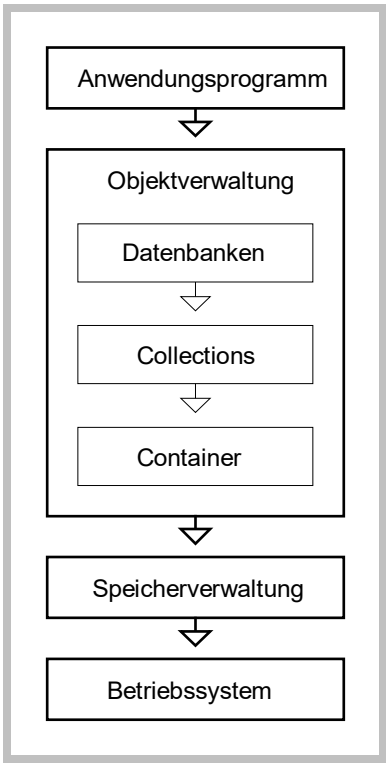

Abb. 2-15: *Verwaltungshierarchie eines Programms*

2.4.1 Container

Container sind für den Einsatz an performancekritischen Stellen gedacht. Bei allen Fragen des Designs und der Implementierung versuchen wir, eine Lösung mit bestmöglicher Performance zu finden. Mengen, die nur einen geringen Einfluß auf die Performance ausüben, können wir mit den komfortableren Collections darstellen. Die Analyse von OHelp ergab bei Collections die folgenden Performancemängel:

- Virtuelle Methoden verlangsamen das Iterieren und den Zugriff.
- Die verwalteten Objekte müssen von einer abstrakten Basisklasse erben.
- Homogene Collections belasten die Speicherverwaltung unnötig.

Das erste Problem beheben wir, indem wir für Container keine abstrakte Basisklasse deklarieren. Dadurch entfallen virtuelle Methoden, die in abgeleiteten Klassen (konkreten Containern) redefiniert werden müssen. Wir fordern jedoch, daß alle Container ein einheitliches Interface besitzen. Damit vereinfacht sich ihre Handhabung. Im Abschnitt 1.2.2

"Überblick ist Alles" haben wir festgestellt, daß viele Klassen mit demselben Interface genauso einfach wie eine einzelne zu bedienen sind. Ein breites Spektrum solcher Klassen bietet hohe Flexibilität. Während der Implementierung kann ein Container durch einen anderen ersetzt werden, ohne daß seine Verwendungen davon betroffen sind.

Das zweite Problem lösen wir mit den sprachlichen Mitteln von C++. Ein Container ist eine Menge gleichartiger Objekte. Wir fordern nicht, daß sie von einer bestimmten Basisklasse erben. Der Container soll sich an seine Objekte anpassen. Eine normale Klasse besitzt diese Fähigkeit nicht. In der Sprache C++ können wir jedoch parametrisierte Datentypen mit Hilfe von Templates darstellen. Einen neuen Container entwerfen wir also in Form eines Klassentemplates. Der Elementtyp ist ein Parameter des Containers.

Eine Collection enthält genau genommen keine Objekte. Sie werden außerhalb erzeugt und gelöscht. Die Collection verwaltet nur Zeiger darauf. Das ist die Ursache des dritten Problems. Die vielen einzelnen Objekte belasten die Speicherverwaltung. In einem Container ist es leicht möglich, den Speicher zu optimieren. Er ist eine homogene Menge und kennt die Größe seiner Objekte. Wir fordern also, daß ein Container den Speicher für seine Elemente selbst verwaltet. Er baut direkt auf der Speicherverwaltung auf und ist enger mit ihr verbunden als eine Collection (siehe Abbildung 2-16).

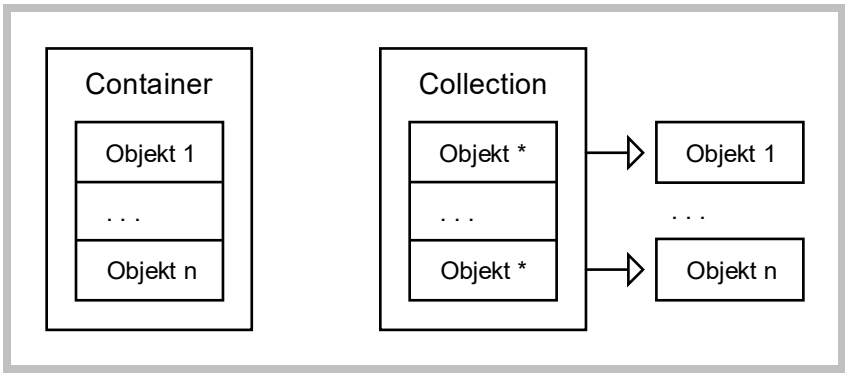

Abb. 2-16: *Container und Collection*

Beim Entwurf der Schnittstelle für Container orientieren wir uns an den Collections von OHelp. Die Datentypen für die Länge und zur Identifizierung eines Eintrags werden nicht global definiert. Sie können sich von Container zu Container unterscheiden und sind als geschachtelte Typen in der Klasse enthalten. Den Begriff EntryId ersetzen wir durch "logischer Zeiger". Auch der Objekttyp ist nicht global vorgegeben. Er gelangt als dritte Typdefinition in die Containerklasse. Die Methodennamen übernehmen wir im wesentlichen von den Collections. Die Methoden zum Einfügen und Löschen nennen wir jedoch geringfügig anders, denn der Container verwaltet den Speicher seiner Objekte selbst (siehe Beispiel weiter unten). `Add` nennen wir nun `AddObj`, `Delete` `DelObj` usw. Sehen wir uns nun die vollständige Deklaration eines Containertemplates an.

```
template <class t_obj>
  class gct_AnyContainer
    {
  public:
    typedef unsigned short t_Length;
    typedef void *       t_Pointer;
    typedef t_obj        t_Object;

    t_Length             GetLen () const;
    t_Pointer            First () const;
    t_Pointer            Next (t_Pointer o_ptr) const;
    t_Object *           GetObj (t_Pointer o_ptr) const;
    t_Pointer            AddObj (const t_Object * po_obj = 0);
    t_Pointer            AddObjCond (const t_Object * po_obj);
```

```
    t_Pointer            AddObjAfter (t_Pointer o_ptr,
                           const t_Object * po_obj = 0);
    t_Pointer            DelObj (t_Pointer o_ptr);
    };
```

Der größte Unterschied zwischen einem Container und einer Collection besteht in der Bereitstellung des Speichers für die Objekte. Sie befinden sich physisch im Container, und er ruft auch ihre Konstruktoren und Destruktoren auf. Ein neues Objekt wird im Container mit seinem Standard- oder Kopier-Konstruktor erzeugt. Der Parameter der Methode `AddObj` hat den Vorgabewert Null. Ein Aufruf der Methode ohne Parameter führt zur Standardinitialisierung des neuen Objekts. Ein vorhandenes Objekt kann nicht in den Container übernommen, sondern nur kopiert werden. Dazu muß der Methode `AddObj` ein Zeiger auf das zu kopierende Objekt übergeben werden. Das neue Objekt wird dann mit dem Kopier-Konstruktor erzeugt.

Die Methode `DelObj` verwaltet ebenfalls das referenzierte Objekt und den zugehörigen Speicher. Sie ruft zuerst den Destrukor des Objekts auf. Dann wird die Verbindung zum Container gelöscht und der belegte Speicher freigegeben. Das folgende Programmfragment enthält je ein Beispiel für das Erzeugen, Kopieren und Löschen eines Objekts in einem Container.

```
ct_String co_string = "Vorhandener Text";
ct_String * pco_string;
gct_AnyContainer <ct_String> co_container;
gct_AnyContainer <ct_String>:: t_Pointer o_ptr;

// Neues Objekt im Container erzeugen und initialisieren
o_ptr = co_container. AddObj ();
pco_string = co_container. GetObj (o_ptr);
(* pco_string) = "Neuer Text";

// Vorhandenes Objekt in den Container kopieren
o_ptr = co_container. AddObj (& co_string);

// Objekt aus dem Container nehmen und löschen
co_container. DelObj (o_ptr);
```

Auch die Methode `AddObjAfter` besitzt einen Parameter mit dem Vorgabewert Null. Sie dient dem positionierten Einfügen eines Objekts in den Container. Ein Aufruf ohne Objektparameter führt zum Erzeugen eines neuen Objekts mit dem Standard-Konstruktor. Der Methode `AddObjCond` muß stets ein Zeiger auf ein Objekt übergeben werden. Sie prüft mit dem Gleich-Operator, ob das angegebene Objekt schon im Container enthalten ist. Wird ein gleiches Objekt gefunden, liefert sie den logischen Zeiger darauf. Andernfalls wird ein neues Objekt mit dem Kopier-Konstruktor erzeugt und dessen logischer Zeiger zurückgegeben.

Eine Übergangsform zwischen Container und Collection ist ein Zeiger-Container, also ein Container, der Zeiger enthält. Dabei nutzen wir die Effizienz des Containers und die Allgemeinheit der Collection. Dieser Container besitzt keine virtuellen Methoden und kann Verweise auf Objekte abgeleiteter Klassen enthalten. Bei der Arbeit mit einem Zeigercontainer müssen wir darauf achten, daß die Methode `AddObj` einen Zeiger auf einen Zeiger erwartet und `GetObj` einen Zeiger auf einen Zeiger liefert. Die eigentlichen Objekte müssen wie bei einer Collection außerhalb erzeugt und gelöscht werden.

```
ct_String co_string = "Vorhandener Text";
ct_String * pco_string;
gct_AnyContainer <ct_String *> co_container;
gct_AnyContainer <ct_String *>:: t_Pointer o_ptr;

// Neues Objekt erzeugen und in den Container aufnehmen
pco_string = new ct_String ("Neuer Text");
```

```
o_ptr = co_container. AddObj (& pco_string);

// Vorhandenes Objekt in den Container kopieren
pco_string = new ct_String (co_string);
o_ptr = co_container. AddObj (& pco_string);

// Objekt aus dem Container nehmen und löschen
pco_string = * co_container. GetObj (o_ptr);
co_container. DelObj (o_ptr);
delete pco_string;
```

Eine Containerklasse besitzt zahlreiche Ähnlichkeiten mit einer Storeklasse. Beide sind auf gute Performance ausgelegt und erben nicht von einer abstrakten Basisklasse mit virtuellen Methoden. Den Datentypen `t_Size` und `t_Pointer` eines Stores entsprechen im Container `t_Length` und `t_Pointer`. Ein Container verwaltet keinen rohen Speicher, sondern Objekte. Deshalb enthält er als dritten Datentyp `t_Object`. Der Storemethode `AddrOf` entspricht die Containermethode `GetObj`. Während `AddrOf` einen untypisierten Zeiger auf einen Speicherbereich liefert (`void *`), erhalten wir von `GetObj` einen typisierten Zeiger auf das referenzierte Objekt. Den reinen Speichermethoden `Alloc` und `Free` ähneln die objektbezogenen Methoden `AddObj` und `DelObj`. Sie verwalten nicht nur den Speicher, sondern erzeugen bzw. löschen auch das darin befindliche Objekt.

Von einem Store fordern wir nicht, daß die Speicherbereiche ihre physische Adresse behalten. Das gilt auch für die Adressen der Objekte in einem Container. Erst dadurch wird es möglich, den Speicher effizient zu verwalten. Nach dem Einfügen oder Löschen eines Objekts kann der Container andere Objekte im Speicher verschieben. In einem Arraycontainer verlieren auch die logischen Zeiger ihre Gültigkeit. Ein Listencontainer stellt sicher, daß der logische Zeiger, den wir beim Einfügen erhalten, bis zum Löschen dasselbe Objekt identifiziert.

Bei der Arbeit mit Containern werden unterschiedliche Anforderungen an die Gültigkeit von Verweisen gestellt. Machmal müssen die logischen Zeiger ihre Gültigkeit behalten, manchmal die Adressen der Objekte. Ein Zeigercontainer stellt sicher, daß die referenzierten Objekte im Speicher an derselben Stelle bleiben. In einem Listencontainer bleiben auch die logischen Zeiger gültig. In Tabelle 2-2 sehen wir die Kombinationen, die sich daraus ergeben. Die Sternchen geben Auskunft über die Effizienz des Containertyps. Viele Sternchen stehen für eine gute Performance.

	Logischer Zeiger ungültig	Logischer Zeiger gültig
Adresse ungültig	Objekt-Array (****)	Objekt-Liste (***)
Adresse gültig	Zeiger-Array (**)	Zeiger-Liste (*)

Tab. 2-2: *Logische Zeiger und Adressen in Containern*

2.4.2 Collections

Das Collection-Konzept übernehmen wir im wesentlichen aus dem ersten Teil des Buchs. Die Namensgebung passen wir jedoch an die Stores und Container an. Um Verwechslungen mit den lokalen Typen der Container zu vermeiden, nennen wir die beiden globalen Typen der Collections `t_CollLen` und `t_CollPtr`. Die Methoden zum Einfügen und Löschen von Objekten heißen nun `AddPtr` und `DelPtr`. Eine Collection verwaltet keine Objekte, sondern nur Zeiger darauf. Werden Container und Collections gleichzeitig eingesetzt, dürfen die Methoden `AddObj` und `AddPtr` nicht miteinander verwechselt werden. Der folgende Programmausschnitt zeigt die neue Deklaration der abstrakten Basisklasse `ct_Collection`.

```
typedef unsigned long t_CollLen;
typedef unsigned long t_CollPtr;

class ct_Collection: public ct_Object
  {
public:
  virtual t_CollLen    GetLen () const = 0;
  virtual t_CollPtr    First () const = 0;
  virtual t_CollPtr    Next (t_CollPtr o_ptr) const = 0;
  virtual ct_Object *  GetObj (t_CollPtr o_ptr) const = 0;
  virtual t_CollPtr    AddPtr (ct_Object * pco_obj) = 0;
  virtual t_CollPtr    AddPtrCond (ct_Object * pco_obj) = 0;
  virtual t_CollPtr    AddPtrAfter (t_CollPtr o_ptr,
                         ct_Object * pco_obj) = 0;
  virtual t_CollPtr    DelPtr (t_CollPtr o_ptr) = 0;
  };
```

Die Methoden der Collections müssen nicht mehr einzeln programmiert werden. Für die Implementierung einer konkreten Collection nutzen wir einen entsprechenden Zeigercontainer, zum Beispiel einen Arraycontainer für eine Arraycollection. In der Anwendung verhalten sich Collections ähnlich wie Zeigercontainer. Die referenzierten Objekte müssen außerhalb erzeugt und gelöscht werden. Das folgende Programmfragment enthält je ein Beispiel für das Erzeugen, Kopieren und Löschen eines Objekts in einer Collection.

```
ct_String co_string = "Vorhandener Text";
ct_String * pco_string;
ct_Collection * pco_collection = ....;
t_CollPtr o_ptr;

// Neues Objekt erzeugen und in die Collection aufnehmen
pco_string = new ct_String ("Neuer Text");
o_ptr = pco_collection-> AddPtr (pco_string);

// Vorhandenes Objekt in die Collection kopieren
pco_string = new ct_String (co_string);
o_ptr = pco_collection-> AddPtr (pco_string);

// Objekt aus der Collection nehmen und löschen
pco_string = pco_collection-> GetObj (o_ptr);
pco_collection-> DelPtr (o_ptr);
delete pco_string;
```

2.5 Sicherheitstraining

2.5.1 Reservespeicher

Moderne Betriebssysteme verfügen über einen virtuellen Speicher. Der Arbeitsspeicher des Computers wird mit Hilfe von Auslagerungsdateien vergrößert. Damit steht einem Anwendungsprogramm ein Vielfaches des Hauptspeichers zur Verfügung. Aber auch der virtuelle Speicher geht einmal zu Ende. Unsere Ansprüche wachsen schneller als die Hardware. Das Multitasking ermöglicht es, mehrere Anwendungen hintereinander zu starten. Das Schließen vergessen wir meist und werden erst daran erinnert, wenn das Betriebssystem sagt: Out of memory.

Als Softwareentwickler müssen wir an jeder Stelle des Programms damit rechnen, daß kein freier Speicher mehr zur Verfügung steht. Unterlassen wir diese Prüfungen, kann unser

Programm unkontrolliert verlassen werden. Das ist meist mit ärgerlichem Datenverlust verbunden. Die traditionelle Lösung des Problems besteht in der Prüfung jeder einzelnen Speicheranforderung. Kann kein Speicher bereitgestellt werden, erhalten wir von den Standardfunktionen `malloc` und `realloc` einen Nullzeiger. Diese Konvention gilt auch für die Methoden `Alloc` und `Realloc` der Storeklassen.

Eine typische Situation für einen Speicherüberlauf ist das Laden einer Textdatei. Betrachten wir dazu ein Programmbeispiel. Die Klasse `ct_TextFile` enthält alle Definitionen zum Einlesen einer Datei. Andere Eigenschaften werden hier nicht berücksichtigt. Der Text wird intern als Folge von Zeilen dargestellt. Für eine Zeile verwenden wir die Klasse `ct_String`, für die Folge die Collection `ct_DList`. Die Methode `Read` liest den gesamten Text ein und setzt eine Statusvariable. Tritt dabei ein Fehler auf, wird der halb eingelesene Text mit der Methode `ClearList` gelöscht.

```
class ct_TextFile
  {
public:
  enum et_ErrorModes
    {
    ec_Ok,
    ec_OutOfMemory,
    ....
    };
private:
  ct_DList *         pco_TextLines;
  et_ErrorModes      eo_ErrorMode;

  void               ClearList ();
  et_ErrorModes      ReadLine (FILE * pco_file, t_CollPtr & ro_ptr);
public:
  void               Read ();
  ....
  };

void ct_TextFile:: Read ()
  {
  eo_ErrorMode = ec_Ok;
  t_CollPtr o_ptr = 0;
  FILE * pco_file;
  .... // Datei öffnen
  while (! feof (pco_file))
    {
    eo_ErrorMode = ReadLine (pco_file, o_ptr);
    if (eo_ErrorMode != ec_Ok)
      {
      ClearList ();
      break;
      }
    }
  .... // Datei schließen
  }
```

Das Einlesen einer einzelnen Zeile erfolgt mit der Methode `ReadLine`. Ihre Parameter sind ein Zeiger auf die geöffnete Datei und eine Referenz auf einen logischen Zeiger. Dieser enthält den Zeigerwert der zuletzt gelesenen Zeile und bekommt den neuen Zeigerwert zugewiesen. Eine neue Zeile benötigt drei Speicherblöcke, je einen für das Stringobjekt, die eigentliche Zeichenkette und das DList-Node. In allen drei Fällen kann der Speicher ausgehen. Wurde das Stringobjekt erzeugt, aber der Speicher für die Zeichenkette fehlt, muß das Objekt vor der `return`-Anweisung gelöscht werden. Dasselbe gilt, wenn die Liste kein neues Node bereitgestellt hat. Die Methode `ClearList` kann nur Stringobjekte löschen, die schon in der Liste enthalten sind.

```
ct_TextFile:: et_ErrorModes
  ct_TextFile:: ReadLine (FILE * pco_file, t_CollPtr & ro_ptr)
  {
  static char ac_buffer [BUFFER_SIZE];
  .... // Neue Zeile aus Datei in ac_buffer lesen
  ct_String * pco_textLine = new ct_String (ac_buffer);
  if (pco_textLine == 0) // Kein Stringobjekt?
    return ec_OutOfMemory;
  if (pco_textLine-> GetStr () == 0) // Keine Zeichenkette?
    {
    delete pco_textLine;
    return ec_OutOfMemory;
    }
  ro_ptr = pco_TextLines-> AddPtrAfter (ro_ptr, pco_textLine);
  if (ro_ptr == 0) // Nicht in die Liste eingefügt?
    {
    delete pco_textLine;
    return ec_OutOfMemory;
    }
  }
```

Der Anweisungsteil der Methode `ReadLine` besteht zum größten Teil aus Fehlerabfragen und deren Behandlung. Für einen korrekten Test der Methode müßte die Speicherverwaltung so manipuliert werden, daß alle drei Fehlersituationen einmal auftreten. Implementierung, Test und Wartung solcher Programmteile sind sehr aufwendig. Deshalb wurde die Ausnahmebehandlung (*Exception Handling*) in den neuen C++-Standard aufgenommen. Damit soll das Reagieren auf Fehler vereinfacht und vereinheitlicht werden. Die Grundidee des Exception Handlings besteht darin, die Fehlerbehandlung vom übrigen Programmcode zu trennen. Eine Verbundanweisung ohne Fehlerbehandlung wird versuchsweise ausgeführt. Tritt dabei ein Fehler auf, wird er von einem separaten Programmteil behandelt. Betrachten wir eine Definition der Methode `ReadLine` mit Exception Handling.

```
ct_TextFile:: et_ErrorModes
  ct_TextFile:: ReadLine (FILE * pco_file, t_CollPtr & ro_ptr)
  {
  try
    {
    static char ac_buffer [BUFFER_SIZE];
    .... // Neue Zeile aus Datei in ac_buffer lesen
    ro_ptr = pco_TextLines-> AddPtrAfter (ro_ptr, new ct_String (ac_buffer));
    }
  catch (xalloc)
    {
    return ec_OutOfMemory;
    }
  return ec_Ok;
  }
```

Der Programmcode ist kürzer und übersichtlicher geworden. Hinter dem Schlüsselwort `try` steht der eigentliche Inhalt der Methode. Dieser ist so implementiert, als könnte kein Fehler auftreten. Daran können sich mehrere `catch`-Blöcke anschließen. Jeder ist für eine bestimmte Fehlerart zuständig. Wir betrachten in unserem Beispiel nur Speichermangelfehler. Diese erzeugen eine Ausnahme (*Exception*) des Typs `xalloc`. Wird sie von der Speicherverwaltung ausgelöst, bricht die Ausführung der aktuellen Anweisung ab, und das Programm wird im zugehörigen `catch`-Block fortgesetzt.

Tritt in einer Verbundanweisung eine Exception auf, werden wie beim normalen Verlassen die Destruktoren lokaler Objekte aufgerufen. Nach einer Exception in einem Konstruktor werden alle vollständig konstruierten Teilobjekte (Attribute und Basisklassen) zerstört. Diese Mechanismen vereinfachen die Aufräumarbeit, sind aber nicht auf dynamisch erzeugte

Objekte anwendbar. In der Methode `ReadLine` wird mit dem Operator `new` ein neues Stringobjekt erzeugt. Kann es wegen Speichermangels nicht in die Liste eingefügt werden, bleibt es nutzlos im Speicher liegen. Wir müssen die Fehlerbehandlung noch einmal überarbeiten.

```
ct_TextFile:: et_ErrorModes
  ct_TextFile:: ReadLine (FILE * pco_file, t_CollPtr & ro_ptr)
  {
  ct_String * pco_textLine = 0;
  try
    {
    static char ac_buffer [BUFFER_SIZE];
    .... // Neue Zeile aus Datei in ac_buffer lesen
    pco_textLine = new ct_String (ac_buffer);
    o_ptr = pco_TextLines-> AddPtrAfter (o_ptr, pco_textLine);
    }
  catch (xalloc)
    {
    delete pco_textLine;
    return ec_OutOfMemory;
    }
  return ec_Ok;
  }
```

Das Exception Handling ist komfortabel wie andere C++-Mechanismen, die wir schon behandelt haben. Wollen wir es einsetzen, müssen wir aufmerksam auf Seiteneffekte achten. Auch hier wartet so manche Falle im Unsichtbaren darauf, daß wir hineintappen. Für die Behandlung von Speichermangelfehlern bietet es keine befriedigende Lösung. Wir müssen immer noch viele Fälle einzeln behandeln. Dynamischer Speicher wird an zahlreichen Stellen des Programms benötigt. Ebenso zahlreich sind die möglichen Fehlersituationen. Wollen wir den Aufwand für deren Behandlung auf ein Minimum reduzieren, müssen wir uns nach einer anderen Technik umsehen.

Optimal wäre es, wenn wir vor einer speicherintensiven Operation prüfen könnten, ob noch genügend freier Speicher vorhanden ist. Die Größe des verfügbaren Speichers wird uns jedoch von den meisten Betriebssystemen nicht mitgeteilt, oder wir erhalten nur eine grobe Schätzung. Zur Verbesserung des Speichermanagements müssen wir uns selbst etwas einfallen lassen. Um herauszufinden, ob der freie Speicher zu Ende geht, fordern wir beim Programmstart einen großen Block **Reservespeicher** an. Kann eine Speicheranforderung nicht erfüllt werden, geben wir den Reservespeicher frei und versuchen es erneut. Das Programm kann solange weiterarbeiten, bis auch der Reservespeicher aufgebraucht ist.

Um die Handhabung des Reservespeichers zu vereinfachen, integrieren wir ihn in eine Storeklasse. Kann der Store keinen neuen Speicher bereitstellen, gibt er die Reserve frei. Als Anwender des Stores merken wir davon nichts. Die Methoden `Alloc`, `Realloc` und `Free` arbeiten normal weiter. Die neue Abfragemethode `HasReserve` liefert aber den Wert `false`. Das ist für uns das Warnsignal. Nun dürfen wir nicht mehr viel Speicher anfordern. Stattdessen sollten wir solange Speicher freigeben, bis der Reservespeicher wieder verfügbar ist. Nach jeder Freigabe versucht der Store, den Reservespeicher erneut anzufordern. Gelingt es, liefert `HasReserve` wieder den Wert `true`. Entwerfen wir eine Storeklasse mit Reservespeicher, sollten die neuen Attribute und Methoden `static` deklariert werden, damit nicht jede Instanz eine eigene Reserve anlegt.

```
class ct_SafeStore
  {
  static unsigned      u_ReserveLen;
  static void *        pv_Reserve;
public:
  // Allgemeines Store-Interface
  static void          SetReserveLen (unsigned u_resLen);
```

```
  static unsigned      GetReserveLen ();
  static bool          HasReserve ();
  };
```

Mit dieser Technik muß nicht mehr jede einzelne Speicheranforderung geprüft werden. Stattdessen fragen wir punktuell, ob noch Reservespeicher vorhanden ist. Die Häufigkeit der Abfragen muß mit der Größe des Reservespeichers übereinstimmen. Zwischen zwei Abfragen darf nicht mehr Speicher angefordert werden, als wir in Reserve haben. Beschränken wir uns zum Beispiel beim Laden einer Textdatei auf eine Zeilenlänge von 64 K Bytes, ist ein Reservespeicher von 65 K Bytes ausreichend. Die Abfrage muß dann bei jeder einzelnen Zeile erfolgen. Betrachten wir die geänderte Definition der Methode `ReadLine`.

```
ct_TextFile:: et_ErrorModes
  ct_TextFile:: ReadLine (FILE * pco_file, t_CollPtr & ro_ptr)
  {
  static char ac_buffer [BUFFER_SIZE];
  .... // Neue Zeile aus Datei in ac_buffer lesen
  ro_ptr = pco_TextLines-> AddPtrAfter (ro_ptr, new ct_String (ac_buffer));
  if (GetGlobalStore ()-> HasReserve ())
    return ec_Ok;
  else
    return ec_OutOfMemory;
  }
```

Ohne Reservespeicher ist es schwer, bei Speichermangel die Konsistenz der Daten zu gewährleisten. Mit der Storemethode `Realloc` können wir einen dynamischen Block vergrößern. Der Inhalt des kleineren Blocks wird dabei in den größeren kopiert. Der übergebene Zeiger auf den alten Block verliert seine Gültigkeit. Der Rückgabewert der Methode ist der Zeiger auf den neuen Block. Reicht der Speicher zum Vergrößern nicht aus, liefert `Realloc` den Wert Null. Dann ist der alte Block nicht mehr verfügbar, und ein neuer existiert nicht. Die Daten, die sich im Block befanden, sind verlorengegangen.

Ein Store mit Reservespeicher stellt sicher, daß elementare Operationen zu Ende geführt werden können. Reicht der Speicher zum Vergrößern eines Blocks nicht aus, wird der Reservespeicher freigegeben. Dieser sollte groß genug sein. Dann kann der größere Block bereitgestellt und der alte Inhalt übernommen werden. Diese Schritte laufen innerhalb der Methode `Realloc` ab. Für den Anwender der Stores bleiben sie verborgen. Er erhält einen Zeiger auf den vergrößerten Block. Die nächste Abfrage der Methode `HasReserve` liefert aber den Wert `false`.

2.5.2 Referenzzähler und sichere Zeiger

Zeiger, die ins Leere zeigen (*Dangling Pointers*), sind ein Alptraum jedes Softwareentwicklers. Komplex strukturierte Daten enthalten zahlreiche Verweise von Objekten auf andere. Die Konsistenz dieser Daten zu gewährleisten, ist keine leichte Aufgabe. Besonders schwierig ist das Verwalten von Objekten, die viele Abhängigkeiten besitzen. Fallen alte weg oder kommen neue hinzu, verkürzt oder verlängert sich die Lebensdauer. Werden Objekte zu spät gelöscht, kann der Speicher überlaufen. Werden sie zu früh entfernt, entstehen Dangling Pointers.

Bei der Implementierung der Anwendungsklassen von OHelp haben wir **Referenzzähler** kennengelernt. Sie geben Auskunft über die Anzahl der Verweise auf ein Objekt. Existieren keine Verweise mehr, kann es gelöscht werden. Der Referenzzähler der Klasse `ct_Topic` enthält die Anzahl der Hyperlinks auf dieses Thema. Es kann erst gelöscht werden, wenn keine zugehörigen Hyperlinks mehr existieren. In diesem Beispiel sind die Abhängigkeiten einseitig. Ein Hyperlink zeigt auf ein Thema. Das Thema besitzt jedoch keinen Verweis zu den referenzierenden Hyperlinks.

In der Praxis treten oft mehrseitige Abhängigkeiten auf. Auch dafür sind Referenzzähler geeignet. Sie gelten dann sogar als alleiniges Existenzkriterium des Objekts. Wird eine Referenz gelöscht, und der Zähler erreicht den Wert Null, zerstört sich das Objekt selbst. Der Destruktor entfernt Referenzen auf andere Objekte, die sich möglicherweise auch zerstören. Bilden die Referenzen Zyklen, besteht die Gefahr der Entstehung isolierter Inseln. Beim Löschen der letzten Referenz auf einen Zyklus verliert er die Verbindung zur Außenwelt. Die Objekte sind nicht mehr erreichbar. Sie können auch nicht gelöscht werden, denn ihre Referenzzähler sind ungleich Null. In Abbildung 2-17 sehen wir die Entstehung einer isolierten Insel. Die Zahlen in Klammern sind die Referenzzähler.

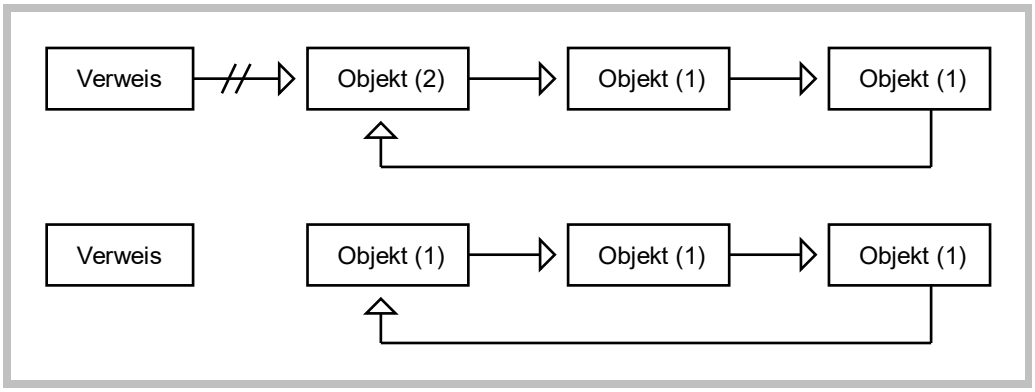

Abb. 2-17: *Insel durch zyklische Referenzzähler*

Referenzzähler in Objekten sind oft eine ungenaue Modellbildung. In vielen Fällen sind sie Zusatzinformationen, die keine Modelleigenschaften widerspiegeln. Ähnlich der Rundung dynamischer Blöcke gehören sie in den Store, der die referenzierten Objekte verwaltet. Eine Speicherverwaltung mit Referenzzählern läßt sich leicht realisieren. Wir sagen dazu kurz **Refstore**. Er ordnet jedem Block einen Referenzzähler zu und besitzt keine eigene Blockverwaltung, sondern baut auf einem anderen Store auf. Dieser kann seine Blöcke dynamisch oder fest verwalten. Eine Refstore-Klasse (z. B. `ct_RefStore`) enthält neben dem allgemeinen Store-Interface die Methoden `IncRef`, `DecRef` und `GetRef`. Diese erwarten als Parameter einen gültigen logischen Zeiger und greifen damit auf den Referenzzähler des Blocks zu.

Wird von einem Refstore neuer Speicher gefordert, beschafft er ihn vom darunter liegenden Store und initialisiert den zugehörigen Referenzzähler. Unabhängig vom Objekt, das im Block untergebracht wird, kann der Referenzzähler erhöht und erniedrigt werden. Der Refstore stellt sicher, daß der logische Zeiger gültig bleibt, solange noch Verweise auf den Speicherblock existieren. Soll der Block freigegeben werden, und der Referenzzähler ist ungleich Null, wird nur ein Flag gesetzt. Dieses kann mit den Methoden `IsAlloc` und `IsFree` abgefragt werden. Ein Zugriff auf den Block ist danach nicht mehr sinnvoll, und die zugehörigen Verweise müssen entfernt werden. Erst wenn der Referenzzähler den Wert Null erreicht, wird der Speicher im anderen Store freigegeben, und der Zeiger verliert seine Gültigkeit (siehe Beispiel weiter unten).

Ein Referenzzähler wird mit der Klasse `ct_RefCount` dargestellt. Diese enthält zwei Informationen. Der eigentliche Referenzzähler ist eine nichtnegative ganze Zahl. Er kann mit den Methoden `IncRef` und `DecRef` geändert und mit `GetRef` abgefragt werden. Das zweite Attribut ist ein Wahrheitswert und benötigt nur ein Bit. Es gibt Auskunft darüber, ob der zugehörige Block im Refstore genutzt oder freigegeben ist. Mit den Methoden `IsAlloc` und `IsFree` kann es abgefragt werden. Die Methode `IsNull` liefert den Wahrheitswert `true`, wenn der Referenzzähler gleich Null und der Block frei ist. Das bedeutet, daß der Block im anderen Store freigegeben werden kann. Abbildung 2-18 zeigt das Interface der Klassen `ct_RefCount` und `ct_RefStore`.

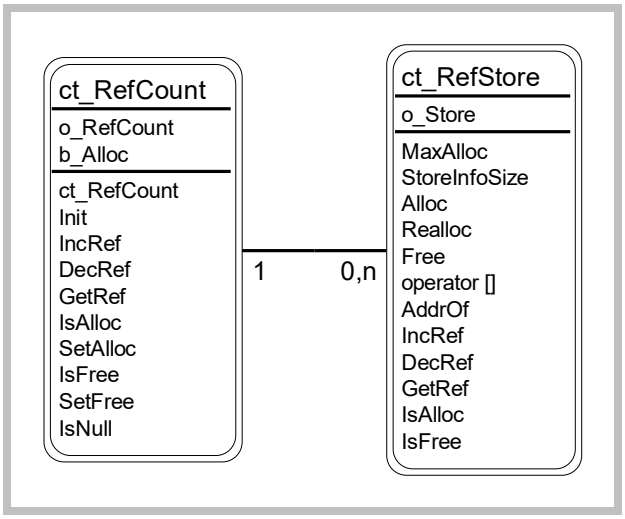

Abb. 2-18: *Referenzzähler und Refstore*

Für die Referenzzähler ist keine eigene Verwaltung nötig. Sie können am Beginn des Blocks, zu dem sie gehören, untergebracht werden. Der Block vergrößert sich dadurch. Beim Zugriff auf den nutzbaren Speicher fragt der Refstore den darunter liegenden Store nach der Adresse und addiert `sizeof (ct_RefCount)` Bytes. Der Zugriff auf den Referenzzähler erfolgt mit der Anfangsadresse des Blocks (siehe Abbildung 2-19). Das `ct_RefCount`-Objekt und die Adreßrechnung sind für den Anwender nicht sichtbar. Sie verbergen sich hinter dem Interface der Klasse `ct_RefStore`.

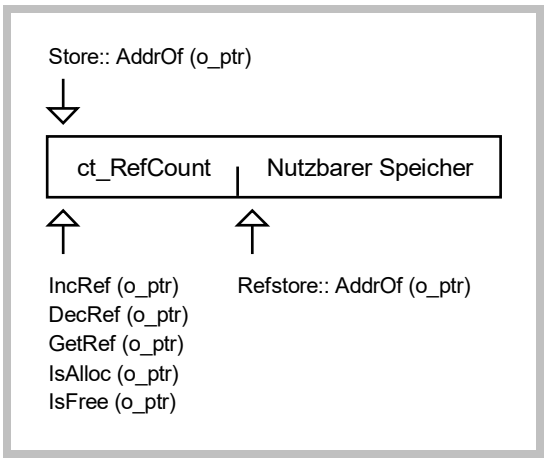

Abb. 2-19: *Ein Block im Refstore*

Das folgende Prinzipbeispiel zeigt die Anwendung eines Refstores. Am Anfang werden drei Objekte definiert, ein Refstore, ein logischer Zeiger und ein C++-Zeiger (Zeilen 3 bis 5). Dann werden vom Refstore 15 Bytes angefordert. Er erhöht die Größe um `sizeof (ct_RefCount)` Bytes und gibt die Anforderung an den darunter liegenden Store weiter. Der logische Zeiger des Blocks wird der Variablen `o_ptr` zugewiesen. Damit der logische Zeiger gültig bleibt, erhöhen wir seinen Referenzzähler (Zeile 7). Wir sehen, daß der Speicher verwendbar ist (Zeile 8), und können seine Adresse berechnen (Zeile 9). In Zeile 10 wird der Speicher im Refstore freigegeben. Der Referenzzähler ist ungleich Null. Deshalb wird nur das Flag `b_Alloc` auf `false` gesetzt. Der logische Zeiger ist noch gültig, aber auf den Speicher kann nicht mehr zugegriffen werden (Zeilen 11 und 12). Beim Verkleinern des Referenzzählers wird die Bedingung `IsNull` erfüllt. Der Refstore gibt nun den Speicher physisch frei (Zeile 13). Damit verliert der logische Zeiger seine Gültigkeit und darf nicht mehr verwendet werden (Zeile 14).

```
 1   void TestRefstore ()
 2     {
 3     ct_RefStore co_refStore;
 4     ct_RefStore:: t_Pointer o_ptr;
 5     void * pv;
 6     o_ptr = co_refStore. Alloc (15);
 7     co_refStore. IncRef (o_ptr);
 8     ASSERT (co_refStore. IsAlloc (o_ptr));
 9     pv = co_refStore. AddrOf (o_ptr);
10     co_refStore. Free (o_ptr);
11     ASSERT (co_refStore. IsFree (o_ptr));
12     // pv = co_refStore. AddrOf (o_ptr);  Nicht mehr erlaubt!
13     co_refStore. DecRef (o_ptr);
14     o_ptr = 0;
15     }
```

Ein Referenzzähler muß stets paarig erhöht und erniedrigt werden. Vergessen wir einmal das Herunterzählen, bleibt er ständig größer als Null, und der zugehörige Speicher kann nicht freigegeben werden. Deshalb verpacken wir den logischen Zeiger in einem Klassenobjekt. Die Zugriffsmethode `SetPtr` und der Destruktor verändern den Referenzzähler, auf den gezeigt wird. Entsteht ein neuer Verweis, wird der zugehörige Referenzzähler erhöht. Beim Löschen eines Verweises wird er um eins erniedrigt. Vor dem Speicherzugriff fragen wir das Zeigerobjekt mit der Methode `CanAccess`, ob es gültig ist. Wir nennen es einen **sicheren Zeiger**, denn es sagt uns, wann es ins Leere zeigt.

Das folgende Programmfragment enthält die Deklaration einer sicheren Zeigerklasse und die Definition dreier wichtiger Methoden. In diesem Beispiel ist der Refstore nicht global bekannt. Deshalb benötigt der sichere Zeiger einen Verweis auf den zugehörigen Refstore. Die Zeigerklasse besitzt statt eines Standard-Konstruktors einen Konstruktor mit der Adresse des Refstore-Objekts. Bei einer sicheren Zeigerklasse dürfen wir nicht vergessen, einen Kopier-Konstruktor, Destruktor und Gleich-Operator zu definieren.

```
class ct_SafePtr
  {
  ct_RefStore *        pco_RefStore;
  ct_RefStore:: t_Pointer o_Ptr;
public:
                       ct_SafePtr (ct_RefStore * pco_refStore);
                       ct_SafePtr (const ct_SafePtr & co_init);
                       ~ct_SafePtr ();
  ct_SafePtr &         operator = (const ct_SafePtr & co_asgn);
  ct_RefStore:: t_Pointer GetPtr () const;
  void                 SetPtr (ct_RefStore:: t_Pointer o_newPtr);
  bool                 CanAccess () const;
  void *               GetAddr () const;
  };

void ct_SafePtr:: SetPtr (ct_RefStore:: t_Pointer o_newPtr)
  {
  if (o_Ptr != o_newPtr)
    {
    if (o_Ptr != 0)
      pco_RefStore-> DecRef (o_Ptr);
    o_Ptr = o_newPtr;
    if (o_Ptr != 0)
      pco_RefStore-> IncRef (o_Ptr);
    }
  }

bool ct_SafePtr:: CanAccess () const
  {
  return (o_Ptr != 0) && pco_RefStore-> IsAlloc (o_Ptr);
```

```
  }

void * ct_SafePtr:: GetAddr () const
  {
  ASSERT (CanAccess ());
  return pco_RefStore-> AddrOf (o_Ptr);
  }
```

2.5.3 Wem gehört was?

Mit Refstores und sicheren Zeigern ist das Problem isolierter Inseln noch nicht gelöst. Aber wir verfügen nun über die technischen Mittel für eine Lösung. Der prinzipielle Fehler besteht darin, Referenzzähler als *alleiniges* Existenzkriterium für Objekte zu betrachten. Aus dem Sachgebiet, das wir modellieren, können meist andere Kriterien abgeleitet werden. Wir versuchen, jedem Objekt einen **Eigentümer** zuzuordnen. Dieser löscht das Objekt, wenn er es nicht mehr benötigt. Andere Objekte, die Verweise darauf besitzen, dürfen es nicht löschen.

Verbindungen mit sicheren Zeigern ermöglichen das Zerstören von Objekten durch ihren Eigentümer. Ein Objekt kann auch gelöscht werden, wenn noch Verweise darauf existieren. Wird beim Prüfen eines Verweises festgestellt, daß er ungültig ist, muß er entfernt werden. Mit dem Entfernen des letzten Verweises wird automatisch der Speicher des gelöschten Objekts freigegeben (siehe Abbildung 2-20).

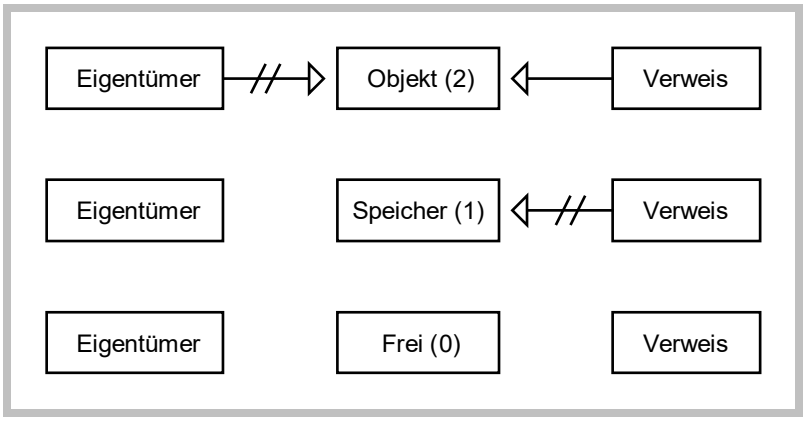

Abb. 2-20: *Löschen eines Objekts durch den Eigentümer*

Enthält ein Objekt abhängige Objekte, müssen die eigenen von den Objekten anderer Eigentümer unterschieden werden. Für das Löschen der eigenen Objekte ist das Objekt selbst zuständig. Wird ein Verweis zu einem anderen Objekt entfernt, darf nur der Referenzzähler erniedrigt werden. Spätestens im Destruktor müssen alle Verweise zu abhängigen Objekten entfernt werden (siehe Abbildung 2-21).

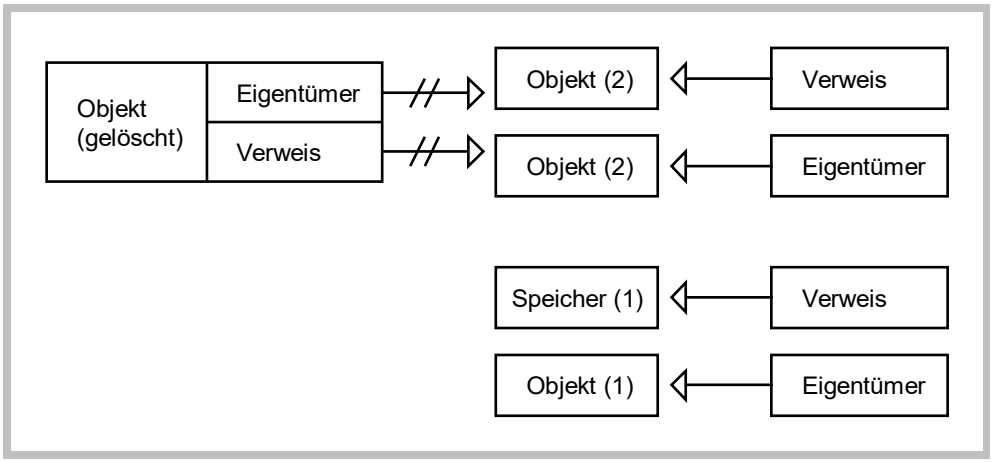

Abb. 2-21: *Entfernen der Verweise im Destruktor*

In einem größeren Programm mit komplex strukturierten Daten kann möglicherweise nicht jedem Objekt ein Eigentümer zugeordnet werden. Wir versuchen, auch diesen Objekten ein Zuhause zu geben, und lassen sie nicht mit losen Verbindungen im Speicher herumirren, sondern übergeben sie einem **globalen Eigentümer**. Globale Objekte werden beim Programmstart automatisch erzeugt und am Programmende zerstört. Sie sind zusammen mit ihren abhängigen Objekten stets erreichbar und sorgen für einen ordnungsgemäßen Programmabschluß.

Objekte müssen nicht während ihrer gesamten Lebensdauer demselben Eigentümer gehören. Ein **Eigentumswechsel** ist keine Gefahr für die Konsistenz unserer Daten. Wir müssen jedoch darauf achten, jeden angefangenen Eigentumsprozeß korrekt zu Ende zu führen. Es ist üblich, für bestimmte Objekttypen Erzeuger- und Entsorgerobjekte zu verwenden. Zum Beispiel ist ein Store sowohl Erzeuger als auch Entsorger für rohe Speicherblöcke. Die Methode `Alloc` liefert einen Zeiger auf einen Block. Dieser geht in das Eigentum des Aufrufers der Methode über. Benötigt er den Speicher nicht mehr, gibt er ihn mit der Methode `Free` an den Store zurück.

Für abhängige Objekte, die schon im statischen Modell eines Programms bekannt sind, bietet die Programmiersprache C++ geeignete Darstellungsformen. Ein eigenes Objekt gelangt als Attribut in die Klasse. Andere Objekte werden mit Zeigern referenziert. Zeiger auf eigene Objekte sollten sparsam verwendet werden. Sie sind zum Beispiel nötig, wenn ein Objekt während des Programmlaufs seinen Eigentümer wechselt. Enthält ein Objekt eine *Menge* abhängiger Objekte, werden sie in einem Container oder einer Collection zusammengefaßt. Ein Container ist nicht nur ein Mittel der Programmoptimierung. Er enthält seine Objekte physisch und sorgt somit für saubere Eigentumsverhältnisse. Von einer Collection können wir nur durch Zusatzwissen ermitteln, ob sie Eigentümer der Objekte ist oder nur Verweise darauf enthält.

Zyklische Verweisketten sind keine Seltenheit. In unserem Beispielprogramm OHelp kann ein Thema einen Verweis auf ein anderes Thema enthalten, das wiederum auf das ursprüngliche verweist. Ein Thema kann programmtechnisch sogar auf sich selbst verweisen, auch wenn der Verweis inhaltlich keinen Sinn ergibt. Bei diesen Verweisen herrschen saubere Eigentumsverhältnisse. Das Thema ist Eigentümer seiner Hyperlinks. Ein Hyperlink ist jedoch kein Eigentümer des Themas, auf das es verweist.

Erlaubt das Design eines Programms eine zyklische Verweiskette, die *nur* aus Eigentümern besteht, können auch isolierte Inseln entstehen (siehe Abbildung 2-22). Dieser Designfehler sollte unbedingt korrigiert werden. Eine einfache Lösung bietet ein globaler Eigentümer. Die Objekte der Verweiskette gehen in seinen Besitz über. Wird der letzte Verweis auf den Zyklus entfernt, sind die Objekte immer noch über ihren Eigentümer erreichbar. Dieser entscheidet, wann die Objekte gelöscht werden.

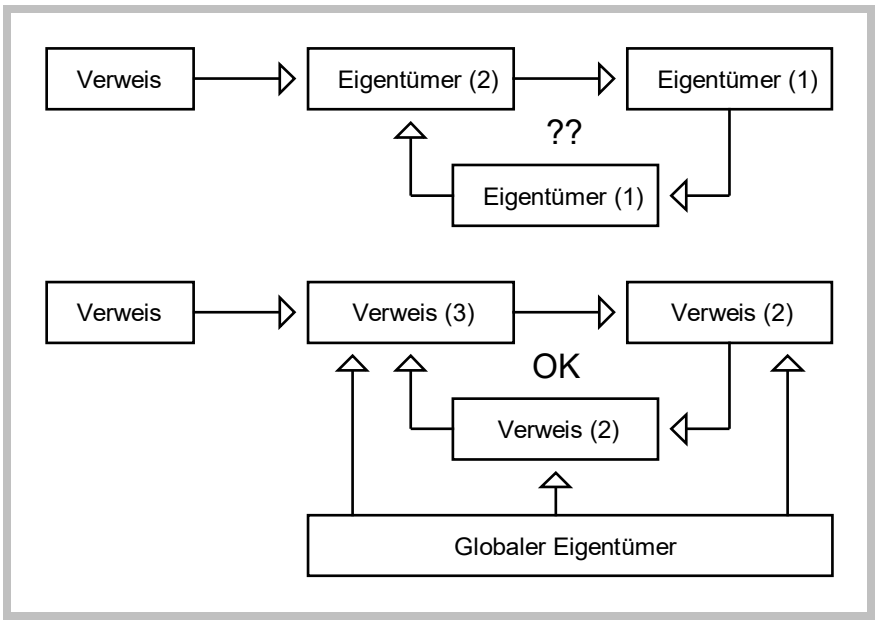

Abb. 2-22: *Korrektur eines Designfehlers*

In Abbildung 2-23 sehen wir die Auflösung einer zyklischen Verweiskette mit sauberen Eigentumsverhältnissen. Der Eigentümer außerhalb der Kette kann zum Beispiel ein Hypertext sein. Im Zyklus befinden sich ein Thema und ein Hyperlink. Der auswärtige Eigentümer ruft den Destruktor des inneren Objekts auf. Dieses zerstört das Verweisobjekt, das ihm gehört. Der Destruktor des Verweisobjekts erniedrigt den Referenzzähler des Speicherblocks, auf den es verwies. Am Ende erreichen beide Referenzzähler den Wert Null, und der belegte Speicher wird freigegeben. Die folgende Regel faßt die wichtigsten Erkenntnisse dieses Abschnitts zusammen.

Refstores und sichere Zeiger ermöglichen das Zerstören von Objekten durch ihren Eigentümer. Attribute und Container sichern saubere Eigentumsverhältnisse. Damit wird die Entstehung isolierter Inseln verhindert.

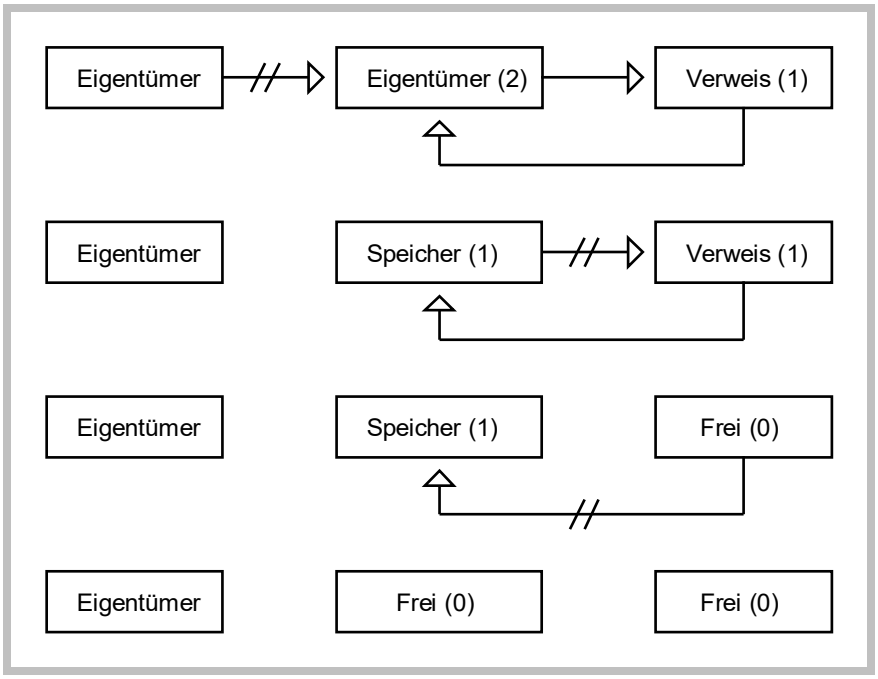

Abb. 2-23: *Auflösung einer Insel*

2.6 Einige Programmiertechniken

2.6.1 Operatoren new und delete

Bei der Implementierung eines Containers stehen wir vor einem technischen Problem: C++-Objekte müssen in selbst verwaltetem Speicher erzeugt und gelöscht werden. Ein Container enthält seine Objekte nicht nur logisch, sondern auch physisch. Er verwaltet ihren Speicher und ruft ihre Konstruktoren und Destruktoren auf. Wird ein neues Objekt in einen Container eingefügt, stellt er den Speicherplatz zur Verfügung und initialisiert es mit dem Standard- oder Kopierkonstruktor. Beim Entfernen eines Objekts wird zuerst der Destruktor aufgerufen und dann der belegte Speicher freigegeben.

In der Programmiersprache C++ existiert nur eine Möglichkeit, ein Objekt in einem selbst verwalteten Speicherblock zu initialisieren, das Definieren eines eigenen Operators `new`. Ein direkter Konstruktoraufruf ist zum Erzeugen eines temporären Objekts erlaubt. Mit einem Zeiger, der auf einen rohen Speicherbereich verweist, kann jedoch kein Konstruktor aufgerufen werden.

```
class ct_AnyClass
  {
public:
            ct_AnyClass ();
  void      AnyMethod ();
  };

// Temp. Objekt erzeugen, AnyMethod aufrufen, temp. Objekt zerstören
ct_AnyClass (). AnyMethod ();

// Speicher bereitstellen und Konstruktor aufrufen
void * pv = malloc (sizeof (ct_AnyClass));
ct_AnyClass * pco = ((ct_AnyClass *) pv)-> ct_AnyClass (); // Kein C++ !!
```

Die Operatoren `new` und `delete` weichen in Syntax und Semantik von anderen Operatoren ab. Deshalb betrachten wir zunächst einen normalen Operator. Er ist eine nichtstatische Methode einer Klasse oder eine globale Funktion. Im folgenden Programmfragment sehen wir eine Klasse mit einem Kleiner-Operator. Er kann auf zwei Arten aufgerufen werden, in der Operatorschreibweise oder wie eine normale Methode der Klasse. Beide Aufrufsformen sind semantisch identisch. Der Compiler fügt beim Aufruf eines Operators keine unsichtbaren Anweisungen hinzu.

```
class ct_AnyClass
  {
public:
  bool      operator < (const ct_AnyClass & co_compare) const;
  } co_Obj1, co_Obj2;

// Aufruf in Operatorschreibweise
if (co_Obj1 < co_Obj2) ....

// Aufruf in Methodenschreibweise
if (co_Obj1. operator < (co_Obj2)) ....
```

Dieselbe Semantik erzielen wir mit einem *globalen* Kleiner-Operator. Er kann auf private Attribute der Klasse nur zugreifen, wenn die Klasse eine entsprechende `friend`-Deklaration enthält. Der Aufruf in Operatorschreibweise gleicht syntaktisch dem vorigen Beispiel. Der direkte Aufruf erfolgt mit seinem Namen und den aktuellen Parametern. Beide Aufrufe stimmen semantisch mit dem Aufruf einer globalen Funktion überein. Auch hier fügt der Compiler keine unsichtbaren Anweisungen hinzu.

```
class ct_AnyClass
  {
  friend bool operator < (const ct_AnyClass &, const ct_AnyClass &);
  } co_Obj1, co_Obj2;

bool operator < (const ct_AnyClass & co1, const ct_AnyClass & co2);

// Aufruf in Operatorschreibweise
if (co_Obj1 < co_Obj2) ....

// Aufruf in Funktionsschreibweise
if (operator < (co_Obj1, co_Obj2)) ....
```

Wir können die Operatoren `new` und `delete` global oder in einer Klasse definieren. Global dürfen sie innerhalb eines Programms nur einmal definiert werden. Wollen wir einer Klasse spezielle Operatoren zuordnen, müssen wir sie als *statische* Methoden in die Klassendeklaration aufnehmen. Sie können keine normalen Methoden sein, denn sie werden auf rohe Speicherbereiche angewendet, die kein gültiges Objekt enthalten. `new` und `delete` werden vom Compiler auch als statisch angesehen, wenn das Schlüsselwort `static` fehlt. Beide Operatoren haben feste Vorgaben für Parameter und Rückgabewerte (siehe folgenden Programmausschnitt). Ein Operator `new` erwartet die Größe des bereitzustellenden Speichers und liefert die Adresse auf den Speicherblock oder Null, wenn kein Speicher zur Verfügung steht. Ein Operator `delete` erwartet die Adresse des gelöschten Objekts.

```
#include <stddef.h> // Für globalen Typ size_t

class ct_AnyClass
  {
public:
  static void *  operator new (size_t u_size);
  static void    operator delete (void * pv);
  };
```

Ein Aufruf von new und delete in Methodenschreibweise gleicht semantisch dem Aufruf anderer statischer Methoden. Der Compiler fügt keine unsichtbaren Anweisungen hinzu. Diese Aufrufe führen nur zum Bereitstellen bzw. Freigeben von Speicher und nicht zum Aufruf von Konstruktoren oder Destruktoren.

```
void * pv = ct_AnyClass:: operator new (15);
ct_AnyClass:: operator delete (pv);
```

Verwenden wir new und delete in Operatorschreibweise, passiert im Unsichtbaren mehr, als auf dem Papier steht. Der Ausdruck new ct_AnyClass entspricht einem Aufruf der unten stehenden Methode CreateNewObject. Zuerst wird die Methode ct_AnyClass:: operator new mit der Größe der Klasse aufgerufen. Konnte der Speicher bereitgestellt werden, wird darin das Objekt initialisiert. Der untypisierte Zeiger wird in einen typisierten umgewandelt und als Ergebnis zurückgegeben.

```
ct_AnyClass * ct_AnyClass:: CreateNewObject ()
  {
  ct_AnyClass * pco =
    (ct_AnyClass *) ct_AnyClass:: operator new (sizeof (ct_AnyClass));
  if (pco != 0)
    pco-> ct_AnyClass (); // Kein C++ !!
  return pco;
  }
```

Die Verwendung von delete in Operatorschreibweise, zum Beispiel delete pco_anyObject, entspricht einem Aufruf der Methode DestroyObject im folgenden Pseudo-C++. Zuerst wird das Objekt mit seinem Destruktor zersört. Dann gibt die Methode operator delete den Speicher frei. Der Compiler stellt sicher, daß Destruktor und delete-Operator zusammenpassen. Besitzt das referenzierte Objekt einen virtuellen Destruktor, werden Destruktor und delete-Operator der tatsächlichen Klasse ermittelt und aufgerufen. Die Besonderheit besteht darin, daß der Operator delete eine statische Methode ist und nicht virtuell sein kann. Der Compiler ermittelt die richtige delete-Methode mit Hilfe des Destruktors.

```
void ct_AnyClass:: DestroyObject (ct_AnyClass * pco)
  {
  if (pco != 0)
    {
    typedef classof (* pco) t_trueClass; // Kein C++ !!
    pco-> ~t_trueClass ();               // Kein C++ !!
    t_trueClass:: operator delete (pco);
    }
  }
```

Mit new und delete können wir auch Arrays von Objekten erzeugen und löschen. Im neuesten C++-Standard existieren dafür die spezialisierten Operatoren new [] und delete []. Bei der Arbeit mit dynamisch erzeugten C++-Arrays müssen wir darauf achten, ein mit new [] erzeugtes Array mit delete [] und nicht mit delete zu löschen. Haben wir vergessen, für eine Klasse die Operatoren new [] und delete [] zu definieren, werden die globalen Äquivalente genutzt, auch wenn die Klasse eigene Operatoren new und delete enthält.

Die Operatormethoden new [] und delete [] besitzen dieselben Parameter und Rückgabewerte wie new und delete. Sie wissen nicht, daß sie den Speicher eines Arrays verwalten und wie groß dieses Array ist. Das bedeutet, daß der Compiler noch mehr im Unsichtbaren tun muß. Wird delete [] in der Operatorschreibweise auf einen typisierten Zeiger angewendet, muß der Compiler die Größe des Arrays aus dem Array selbst ermitteln, um die korrekte Anzahl Destruktoren aufzurufen. Beim Erzeugen eines dynamischen Arrays mit dem Operator new [] muß also die Größe des Arrays im Speicher hinterlegt werden. Sollen n Objekte vom Typ ct_AnyClass erzeugt werden, fordert der Compiler n * sizeof (ct_AnyClass) + sizeof (size_t)

Bytes von der Methode new []. Er speichert die Größe des Arrays und ruft anschließend n mal den Konstruktor der Klasse ct_AnyClass auf.

Von den Sprachdesignern wird empfohlen, auch dynamische C++-Arrays primitiver Datentypen mit dem Operator new [] zu erzeugen. Zum Beispiel sollten wir new char [20] statt malloc (20) verwenden. Zum Verwalten des Speichers werden stets die globalen Operatorfunktionen new, new [], delete und delete [] aufgerufen. Primitive Datentypen besitzen keine Konstruktoren und Destruktoren. Wird ein so erzeugtes Array mit delete [] gelöscht, müssen keine Destruktoren aufgerufen werden. Der Compiler benötigt die implizit gespeicherte Länge des Arrays in diesem Fall nicht. Sie stellt aber einen Speicheroverhead dar. In den unsichtbaren Anweisungen zum Erzeugen eines Arrays mit new [] sollte der Compiler also primitive Datentypen und Klassen unterschiedlich behandeln.

```
class ct_AnyClass
  {
public:
  static void *  operator new (size_t u_size);
  static void *  operator new [] (size_t u_size);
  static void    operator delete (void * pv);
  static void    operator delete [] (void * pv);
  };

ct_AnyClass * pco_Object  = new ct_AnyClass;      // ct_AnyClass:: new
ct_AnyClass * pco_Objects = new ct_AnyClass [10]; // ct_AnyClass:: new []
delete pco_Object;     // ct_AnyClass:: delete
delete [] pco_Objects; // ct_AnyClass:: delete []

char * pc_Char  = new char;      // :: new
char * pc_Chars = new char [10]; // :: new []
delete pc_Char;     // :: delete
delete [] pc_Chars; // :: delete []
```

Der Operator new kann mit zusätzlichen Parametern mehrfach überladen werden. Beim Aufruf in Operatorschreibweise wird der erste, implizite Parameter nicht angegeben, sondern nur die folgenden. Die Parameter des Operators new dürfen nicht mit Konstruktorparametern des zu erzeugenden Objekts verwechselt werden. Ein Überladen des Operators delete ist nicht möglich. Er existiert ähnlich wie ein Destruktor für jede Klasse nur einmal.

```
class ct_AnyClass
  {
public:
                ct_AnyClass ();
                ct_AnyClass (const char * pc);
                ~ct_AnyClass ();
  static void *  operator new (size_t u_size);
  static void *  operator new (size_t u_size, int i);
  static void    operator delete (void * pv);
  };

int i_newParam = 5;
ct_AnyClass * pco = new (i_newParam) ct_AnyClass ("ConstructorParam");
delete pco;
```

Nun verfügen wir über das nötige Detailwissen, um das anfangs gestellte Problem zu lösen. Ein C++-Objekt, zum Beispiel vom Typ ct_AnyClass, soll in selbst bereitgestelltem Speicher erzeugt und gelöscht werden. Die Klasse besitzt möglicherweise eigene Operatoren new und delete. An der Klassendeklaration können wir nachträglich nichts ändern. Deshalb benötigen wir eine Hilfsklasse. Diese enthält als einziges Attribut das zu erzeugende Objekt und als einzige Methoden new und delete. Der Operator new besitzt als zweiten Parameter einen untypisierten C++-Zeiger. Dieser verweist auf den außerhalb bereitgestellten Speicher und wird unverändert zurückgegeben. Der Operator delete hat eine leere Definition. Ein Aufruf in

Operatorschreibweise bewirkt nur das Zerstören des Objekts mit seinem Destruktor. Der Speicher wird außerhalb der `delete`-Methode freigegeben.

Mit dieser Technik können wir Objekte in beliebig bereitgestelltem Speicher erzeugen und löschen. Im folgenden Programmfragment sehen wir die Deklaration eines Klassentemplates. Es erwartet als Parameter den Typ des Objekts. Die Hilfsklasse erbt nicht von der Objektklasse, sondern enthält sie als Attribut. Somit können wir das Template auch auf primitive Datentypen anwenden. Unter der Templatedeklaration sehen wir die Einzelschritte, mit denen ein Objekt erzeugt und gelöscht wird.

```
template <class ct_part>
  class gct_NewDel
    {
    ct_part        co_part;
  public:
    ct_part *      GetObj () { return & co_part; }
    static void *  operator new (size_t, void * pv) { return pv; }
    static void    operator delete (void *) { }
    };

ct_MyStore co_MyStore;

// Speicher anfordern
void * pv_mem =
  co_MyStore. AddrOf (co_MyStore. Alloc (sizeof (ct_AnyClass)));

// Hilfsobjekt erzeugen
gct_NewDel <ct_AnyClass> * pco_obj =
  new (pv_mem) gct_NewDel <ct_AnyClass>;

// Auf das eigentliche Objekt zugreifen
pco_obj-> GetObj ()-> ....;

// Hilfsobjekt zerstören
delete pco_obj;

// Speicher freigeben
co_MyStore. Free (co_MyStore. LogPtrOf (pco_obj));
```

2.6.2 Jungle of Scopes

Store- und Containerklassen enthalten geschachtelte Datentypen, zum Beispiel `t_Pointer`. In C sind alle Typen global. Die Typdefinition innerhalb der Klassen und Methoden ist eine Erweiterung von C++. Das von den Sprachdesignern entworfene Regelsystem für Gültigkeitsbereiche (*Scopes*) ist jedoch an einigen Stellen unübersichtlich. Für eine bessere Orientierung in diesem Regelwerk unternehmen wir nun einen kleinen Spaziergang durch den Jungle of Scopes.

Außerhalb jeder Klasse und Funktion befinden wir uns im Gültigkeitsbereich der Dateien, dem *File Scope*. Besser verständlich ist die Bezeichnung globaler Scope. Alle Namen, die darin deklariert werden, sind global gültig. Er kann die Deklaration von Datentypen, Konstanten, Variablen, Funktionen und Templates enthalten. Die Scopes der globalen Funktionen und Klassen sind dem globalen Gültigkeitsbereich direkt untergeordnet (siehe Abbildung 2-24).

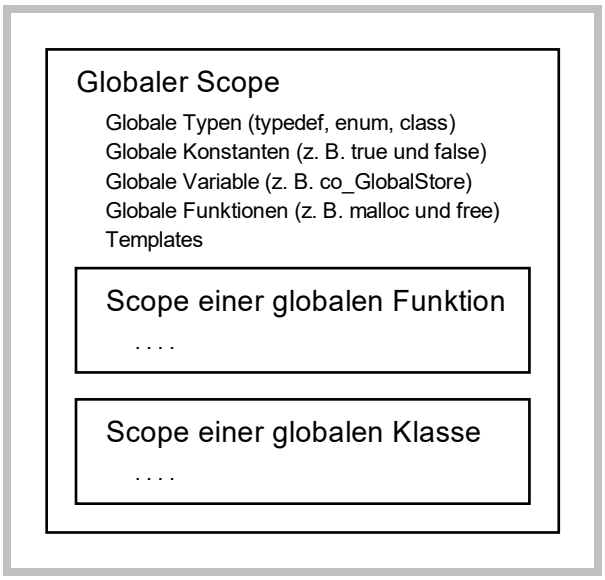

Abb. 2-24: *Globaler Scope*

Die Definition einer globalen Funktion oder einer Methode (Klassen-Funktion) setzt sich aus Verbundanweisungen zusammen. Eine Verbundanweisung wird durch geschweifte Klammern begrenzt und trägt auch die Bezeichnung Block. Der äußerste Block hat eine besondere Bedeutung. Ihm werden die formalen Parameter der Methode zugeordnet. Jede Verbundanweisung bildet einen Scope. Seine genaue Bezeichnung lautet *Local Scope*. Darin können sich Typen, Konstanten, Variable und weitere Verbundanweisungen befinden. Der äußerste Block mit seinen Unterblöcken ergibt den *Function Scope*. Er ist ausschließlich für Sprungmarken relevant. Alle anderen Namen gelten nur innerhalb des Blocks, in dem sie deklariert werden, und seinen Teilblöcken (siehe Abbildung 2-25).

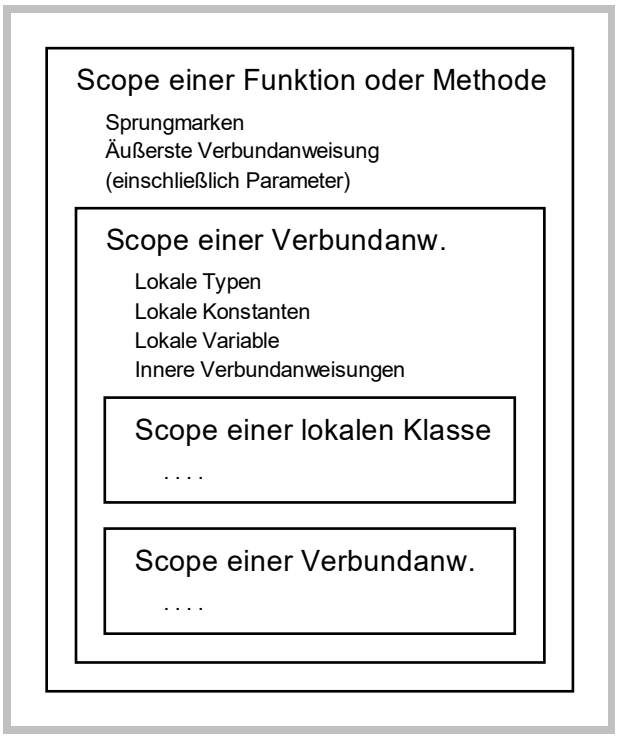

Abb. 2-25: *Methoden- und lokaler Scope*

Auch eine Klasse besitzt einen Scope. Neben Attributen und Methoden können darin Datentypen, zum Beispiel Klassen, deklariert werden. Zur Unterscheidung von lokalen

Klassen in Verbundanweisungen werden sie geschachtelte Klassen (*Nested Classes*) genannt. Sie sind ein Strukturierungsmittel für Gültigkeitsbereiche. Ihre Attribute und Methoden sind kein Bestandteil der äußeren Klasse (siehe Abbildung 2-26).

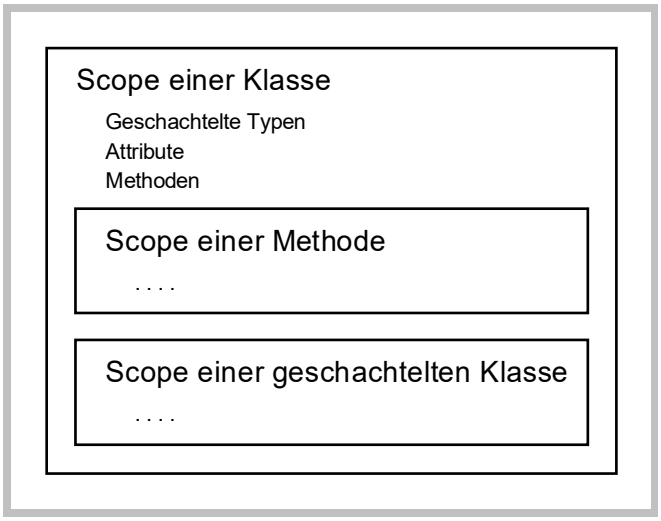

Abb. 2-26: *Klassenscope*

C++ erlaubt das mehrfache Schachteln von Scopes. In einem inneren dürfen Namen des umfassenden Scopes neu definiert werden. Der innerste Scope hat die höchste Priorität. Dort werden Namen zuerst gesucht. Ist ein Name unbekannt, wird die Suche schrittweise nach außen fortgesetzt. Eine Ausnahme bilden abgeleitete Klassen. Das folgende Programmfragment zeigt die Deklaration zweier Klassen mit geschachtelten Datentypen.

```
typedef char t1;
typedef char t2;
typedef char t3;

class ct_Base
  {
public:
  typedef short t1;
  typedef short t2;
  };

class ct_Derived: public ct_Base
  {
public:
  typedef long t1;
  void Method ();
  };

void ct_Derived:: Method ()
  {
  t1 * po1; // Entspricht  long  * po1;  oder  ct_Derived:: t1 * po1;
  t2 * po2; // Entspricht  short * po2;  oder  ct_Base:: t2 * po2;
  t3 * po3; // Entspricht  char  * po3;  oder  ::t3 * po3;
  }
```

Beide Klassen sind von *außen* betrachtet direkt dem globalen Scope untergeordnet. Wäre der Scope der abgeleiteten Klasse der Basisklasse untergeordnet, müßten wir mit `ct_Base:: ct_Derived:: Method` auf die Methode zugreifen. Der Zugriff erfolgt aber mit `ct_Derived:: Method`. Befinden wir uns jedoch *innerhalb* des Scopes der abgeleiteten Klasse, sind wir dem Scope der Basisklasse untergeordnet. Die Suche von Namen erfolgt in der Reihenfolge: Abgeleitete Klasse, Basisklasse, globaler Scope (siehe Abbildung 2-27).

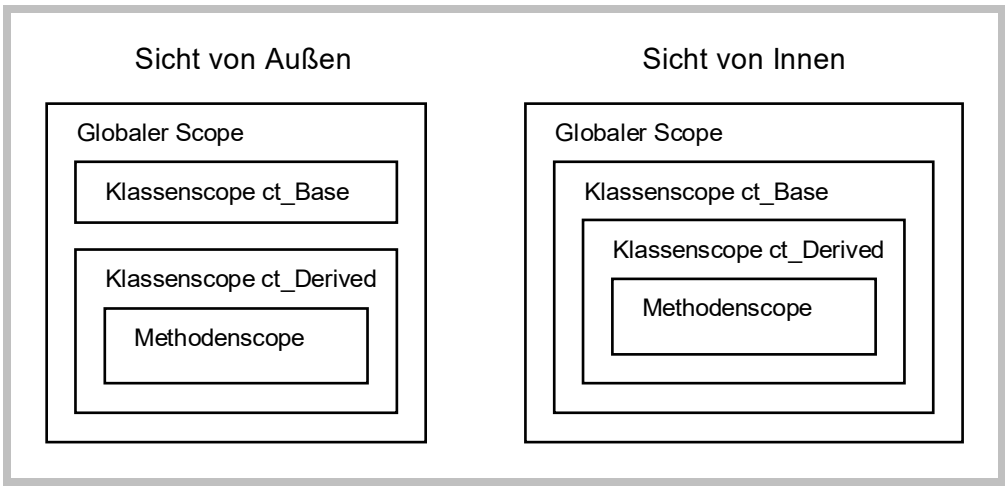

Abb. 2-27: *Äußere und innere Sicht eines Klassenscopes*

Nach diesen allgemeinen Betrachtungen wenden wir uns wieder den Stores und Containern zu. Sie setzen für Parameter und Rückgabewerte ihrer Methoden geschachtelte Typen ein. Wollen wir einen geschachtelten Typ außerhalb der Klasse verwenden, müssen wir seinen Scope angeben, zum Beispiel `ct_AnyStore:: t_Pointer`. Die Definition einer Methode wird entweder in der Klassendeklaration oder im globalen Scope plaziert. Das Inline-Definieren innerhalb der Klassendeklaration wollen wir aber vermeiden, um die Lesbarkeit zu erhöhen.

Die Definition einer Methode beginnt mit dem Rückgabewert. Dieser gehört genau genommen in den Scope der Methode. Der Methodenname ist an dieser Stelle aber noch nicht bekannt. Deshalb muß bei Rückgabewert und Methodenname die zugehörige Klasse angegeben werden. Die Liste der formalen Parameter steht hinter dem Methodennamen. Dort kann der Compiler den korrekten Scope ermitteln. Der Methodenscope ist dem Klassenscope untergeordnet. Deshalb sind geschachtelte Typen in der Parameterliste bekannt und benötigen keine explizite Scopeangabe (siehe Abbildung 2-28).

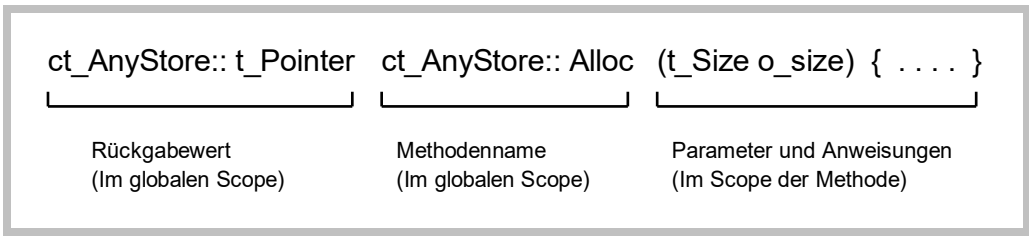

Abb. 2-28: *Scopes in einer Methodendefinition*

Geschachtelte Datentypen erhöhen wesentlich die Flexibilität einer Klassenbibliothek. Wir werden nicht darauf verzichten und gewöhnen uns lieber an die umständliche Methodendefinition. Bei einer globalen Klasse wie in Abbildung 2-28 ist der Schreibaufwand akzeptabel. Etwas mehr müssen wir für ein Klassentemplate investieren. Dort setzt sich der Scope aus dem Templatenamen und den Templateparametern zusammen. Diese Schreibweise ist so unleserlich, daß sogar einige Compiler darüber ins stolpern geraten.

```
template <class t_obj>
  class gct_AnyContainer
    {
  public:
    typedef void *       t_Pointer;
    t_Pointer            Next (t_Pointer o_ptr) const;
    ....
    };
```

```
template <class t_obj>
  gct_AnyContainer <t_obj>:: t_Pointer
  gct_AnyContainer <t_obj>:: Next (t_Pointer o_ptr) const
    {
    ....
    }
```

Der Spaziergang durch den Jungle of Scopes soll nicht ausarten. Für Dschungelfreunde unternehmen wir aber noch einen kleinen Abstecher in die Tiefe. Wollen wir einen Zeiger oder eine Referenz auf eine Klasse deklarieren, benötigen wir nicht ihre vollständige, sondern nur eine Vorwärtsdeklaration. In C++ existieren dafür zwei Möglichkeiten. Eine reine Vorwärtsdeklaration enthält nur das Schlüsselwort (`class`, `struct`, `union`) und den Namen. In einer gemischten wird noch ein anderer Name deklariert oder definiert.

Befinden wir uns in einem lokalen oder Klassenscope, verhält sich eine reine Vorwärtsdeklaration anders als eine gemischte. Eine reine erzeugt stets einen neuen Klassennamen im inneren Scope. Bei einer gemischten Vorwärtsdeklaration wird zuerst geprüft, ob der angegebene Name schon in einem äußeren Scope existiert. Ist er unbekannt, wird er im inneren Scope deklariert. Das folgende Programmfragment zeigt einige Beispiele für Vorwärtsdeklarationen von Klassen.

```
class ct1;        // Reine Vorwärtsdekl. globale Klasse ct1
class ct2 * pco2; // Gem.  Vorwärtsdekl. globale Klasse ct2

class ct_Global
  {
  ct1 * pco1;       // Verwendung globale Klasse ct1
  class ct2 * pco2; // Verwendung globale Klasse ct2
  class ct3 * pco3; // Gem.  Vorwärtsdekl. geschachtelte Klasse ct3
  class ct1;        // Reine Vorwärtsdekl. geschachtelte Klasse ct1
  ct1 * pco4;       // Verwendung geschachtelte Klasse ct1
  :: ct1 * pco5;    // Verwendung globale Klasse ct1
  };
```

Die `friend`-Deklaration einer Klasse hat syntaktische Ähnlichkeit mit einer reinen Vorwärtsdeklaration. Semantisch ähnelt sie einer gemischten. De facto ist sie weder das eine noch das andere. Ist der Name der `friend`-Klasse bekannt, wird die bekannte Klasse wie bei einer gemischten Vorwärtsdeklaration verwendet. Ist der Name noch unbekannt, wird er im umfassenden Scope deklariert (nicht im inneren!). Enthält zum Beispiel eine globale Klasse eine unbekannte `friend`-Deklaration, wird die neue Klasse dem globalen Scope zugeordnet.

```
class ct1; // Reine Vorwärtsdekl. globale Klasse ct1

class ct_Global
  {
  friend class ct1; // Verwendung globale Klasse ct1
  friend class ct2; // Friend-Vorwärtsdekl. globale Klasse ct2
  };
```

3 C++-Bausteine für High-Performance-Programme

Im zweiten Teil des Buchs haben wir zahlreiche Konzepte zur Verbesserung der Performance eines Programms kennengelernt. Der dritte Teil enthält ihre Implementierung. Sie ist ein wichtiger Bestandteil programmtechnischer Konzepte. Das beste Konzept ist wertlos, wenn es nicht sachgerecht in die Praxis umgesetzt wird. C++ bietet mehr Optimierungsmöglichkeiten als andere objektorientierte Sprachen. Wir werden versuchen, viele dieser Möglichkeiten für unser Performance-Tuning einzusetzen.

3.1 Beginn beim Fundament

3.1.1 Dynamische Stores

Das Fundament der Verwaltungshierarchie eines Programms bildet die Speicherverwaltung. Darauf bauen die Objektverwaltung und das eigentliche Anwendungsprogramm auf. Innerhalb der Speicherverwaltung bilden auch die Stores eine Hierarchie. Auf der untersten Stufe stehen ein oder mehrere Stores in direkter Verbindung zur C-Laufzeitbibliothek oder zum Betriebssystem.

Die Speicherverwaltung konkreter Betriebssysteme wollen wir nicht betrachten. Stattdessen nutzen wir die systemunabhängige Schnittstelle der C-Standardbibliothek. Die darauf aufbauende Storeklasse nennen wir `ct_StdStore`. Sie besitzt keine neue Funktionalität, sondern hüllt die Standardfunktionen `malloc`, `realloc` und `free` in ein objektorientiertes Gewand. Leider erfahren wir von der Standardbibliothek nicht, wieviele Bytes konstanten Verwaltungsspeicher (ohne Rundung) sie pro Block benötigt. Wir müssen also vom Durchschnittswert Vier ausgehen (siehe Abschnitt 1.6.3). Die Methoden der Storeklasse sind so einfach, daß wir sie `inline` definieren können. Das folgende Programmfragment enthält die Deklaration der Klasse und die Definition dreier Methoden.

```
class ct_StdStore
  {
public:
  typedef unsigned     t_Size;
  typedef void *       t_Pointer;

  inline unsigned long MaxAlloc () const;
  inline unsigned      StoreInfoSize () const;
  inline t_Pointer     Alloc (t_Size o_size);
  inline t_Pointer     Realloc (t_Pointer o_ptr, t_Size o_size);
  inline void          Free (t_Pointer o_ptr);
  inline void *        AddrOf (t_Pointer o_ptr) const;
  inline t_Pointer     LogPtrOf (void * pv_adr) const;
  };

inline unsigned ct_StdStore:: StoreInfoSize () const
  {
  return 4;
```

```
  }

inline ct_StdStore:: t_Pointer ct_StdStore:: Alloc (t_Size o_size)
  {
  return malloc (o_size);
  }

inline void * ct_StdStore:: AddrOf (t_Pointer o_ptr) const
  {
  return o_ptr;
  }
```

Die Rundung der Blockgröße ist eine erste Erweiterung der Standardfunktionalität. Die neue Storeklasse nennen wir `ct_RndStore`. Sie baut auf dem Standardstore auf und nutzt seine Methoden. Die globale Speicherverwaltung der C-Standardbibliothek ist in einem Programm nur einmal enthalten. Deshalb nehmen wir die Klasse `ct_StdStore` als statisches Attribut in den Roundstore auf. Für die Rundung verwenden wir ein Verfahren mit Minimalgröße und variabler Schrittlänge (siehe Abschnitt 2.3.2). Die Schrittlänge wird über einen Schritt-Teiler gesteuert. Der Roundstore verfügt weiterhin über einen Reservespeicher variabler Größe (siehe Abschnitt 2.5.1). Im folgenden Programmausschnitt sehen wir die vollständige Deklaration der Klasse `ct_RndStore`.

```
class ct_RndStore
  {
public:
  typedef ct_StdStore:: t_Size    t_Size;
  typedef ct_StdStore:: t_Pointer t_Pointer;
private:
  static t_Size        o_ReserveLen; // Länge des Reservespeichers
  static t_Pointer     o_ReservePtr; // Zeiger auf den Reservespeicher
  static ct_StdStore   co_Store;     // Statischer Standardstore
  t_Size               o_MinSize;    // Minimalgröße
  unsigned             u_StepDiv;    // Schritt-Teiler
  unsigned             u_StepDivLog; // Hilfsgröße

  t_Size               Round (t_Size o_size);
public:
                       ct_RndStore ();
  unsigned long        MaxAlloc () const;
  unsigned             StoreInfoSize () const;
  t_Pointer            Alloc (t_Size o_size);
  t_Pointer            Realloc (t_Pointer o_ptr, t_Size o_size);
  void                 Free (t_Pointer o_ptr);
  inline void *        AddrOf (t_Pointer o_ptr) const;
  inline t_Pointer     LogPtrOf (void * pv_adr) const;
  static void          SetReserveLen (t_Size o_resLen);
  static t_Size        GetReserveLen ();
  static bool          HasReserve ();
  void                 SetMinSize (t_Size o_minSize);
  inline t_Size        GetMinSize () const;
  void                 SetStepDiv (unsigned u_stepDiv);
  inline unsigned      GetStepDiv () const;
  };
```

Der Roundstore besitzt keine eigene Speicherverwaltung. Er wirkt wie ein Nachbrenner zu einer vorhandenen. Zum Verändern der Größe eines Blocks mit der Methode `Realloc` wird die ursprüngliche Größe benötigt. Der darunter liegende Store gibt aber keine Auskunft über die Größe seiner Blöcke. Deshalb muß der Roundstore zu jedem Block die gerundete Größe speichern. Er verfährt dabei ähnlich wie ein Refstore. Der Block wird um `sizeof (t_Size)` Bytes vergrößert und die gerundete Größe am Anfang des Blocks untergebracht.

Die Methode `AddrOf` eines Refstores berechnet die Speicheradresse eines logischen Zeigers. Dabei werden zur Blockadresse des darunter liegenden Stores `sizeof (ct_RefCount)` Bytes addiert (siehe die Abschnitte 2.5.2 und 3.2.3). Beim Roundstore können wir diese Adreßrechnung vereinfachen. Der Zeigertyp des darunterliegenden Stores ist bekannt. Der Standardstore verwendet untypisierte C++-Zeiger. Dazu können wir schon beim Anfordern des Speichers `sizeof (t_Size)` Bytes addieren. Die Methode `Alloc` des Roundstores liefert also einen untypisierten C++-Zeiger auf den nutzbaren Bereich des Blocks. `Realloc` und `Free` verarbeiten ebenfalls diese Zeiger (siehe Abbildung 3-1). In der Methode `AddrOf` ist keine Adreßrechnung mehr nötig. Damit beschleunigt sich der Zugriff auf den vom Roundstore verwalteten Speicher.

```
inline void * ct_RndStore:: AddrOf (t_Pointer o_ptr) const
  {
  return o_ptr;
  }
```

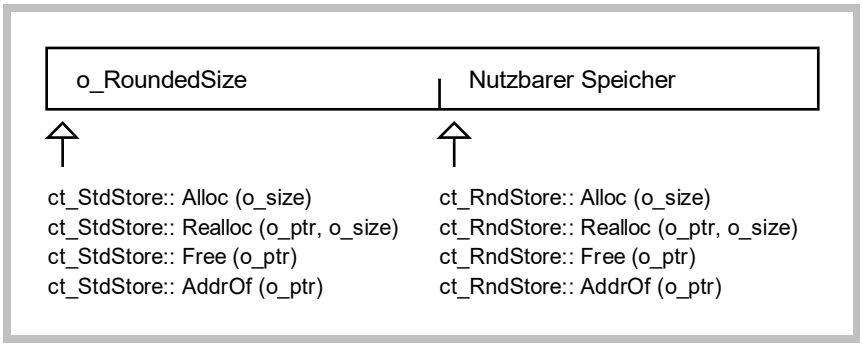

Abb. 3-1: *Logische Zeiger im Roundstore*

Mehrere Methoden des Roundstores sind an der Verwaltung des Reservespeichers beteiligt. Betrachten wir als Beispiel die Implementierung von `Realloc`. Nach Überprüfung der Spezialfälle `o_ptr == 0` und `o_size == 0` wird der Zeiger auf die alte, gerundete Größe ermittelt (`po_oldSize`). Anschließend berechnen wir die neue gerundete Größe (`o_newSize`) und vergleichen sie mit der alten. Sind beide gleich, befindet sich die Größenänderung innerhalb der Schrittweite, und wir geben den unsprünglichen Zeiger zurück. Andernfalls versuchen wir, vom Standardstore einen Block der neuen, gerundeten Größe anzufordern. Gelingt es nicht, geben wir den Reservespeicher frei und versuchen es erneut. Steht nun der neue Speicher zur Verfügung, kopieren wir den Inhalt des alten Blocks in den neuen und tragen an dessen Beginn die gerundete Größe ein. Abschließend wird der alte Block freigegeben und der Rückgabewert berechnet.

```
ct_RndStore:: t_Pointer
ct_RndStore:: Realloc (t_Pointer o_ptr, t_Size o_size)
  {
  if (o_ptr == 0)
    return Alloc (o_size);
  else
    if (o_size == 0)
      {
      Free (o_ptr);
      return 0;
      }
    else
      {
      t_Size * po_oldSize = (t_Size *) o_ptr - 1;
      t_Size o_newSize = Round (o_size);
      if (o_newSize == * po_oldSize)
        return o_ptr;
      else
        {
```

```
        t_Size * po_newSize = (t_Size *) co_Store. Alloc (o_newSize);
        if ((po_newSize == 0) && (o_ReservePtr != 0))
          {
          co_Store. Free (o_ReservePtr);
          o_ReservePtr = 0;
          po_newSize = (t_Size *) co_Store. Alloc (o_newSize);
          }
        if (po_newSize != 0)
          {
          memcpy (po_newSize, po_oldSize,
            o_newSize < * po_oldSize ? o_newSize : * po_oldSize);
          * po_newSize = o_newSize;
          }
        Free (o_ptr);
        return po_newSize != 0 ? po_newSize + 1 : 0;
        }
      }
  }
```

Die private Methode `Round` berechnet zu einer gegebenen Blockgröße den gerundeten Wert. Sie wird bei jedem Aufruf der Methoden `Alloc` und `Realloc` verwendet, ist also zeitkritisch. Das Attribut `u_StepDivLog` dient der Beschleunigung der Berechnung. Es enthält den um eins vergrößerten Zweierlogarithmus des Schritt-Teilers `u_StepDiv`. In unserem Rundungsalgorithmus müssen Minimalgröße und Schritt-Teiler Zweierpotenzen sein. Diese Bedingung prüfen wir mit der Formel `((X & - X) == X)`. Sie beruht auf der Eigenschaft moderner Computer, negative Zahlen im Basiskomplement und nicht im Zweierkomplement darzustellen. Die Methode `SetStepDiv` stellt weiterhin sicher, daß der Schritt-Teiler nicht größer als die Minimalgröße werden kann.

```
void ct_RndStore:: SetStepDiv (unsigned u_stepDiv)
  {
  ASSERT (u_stepDiv != 0);
  ASSERT ((u_stepDiv & - u_stepDiv) == u_stepDiv);
  u_StepDiv = u_stepDiv;
  if (u_StepDiv > o_MinSize)
    u_StepDiv = o_MinSize;
  unsigned u = u_StepDiv;
  u_StepDivLog = 0;
  while (u > 0)
    {
    u >>= 1;
    u_StepDivLog ++;
    }
  }
```

Beim Runden müssen wir die `StoreInfoSize` des Standardstores berücksichtigen. Am Anfang addieren wir sie und `sizeof (t_Size)` zur Blockgröße. Dann vergleichen wir die Blockgröße mit der Minimalgröße. Ist sie kleiner, geben wir die um die `StoreInfoSize` verminderte Minimalgröße zurück. Andernfalls berechnen wir die Schrittweite `o_stepWidth`. Dazu verwenden wir die Hilfsvariable `o_shiftedSize`. Sie ist gleich der durch `2 * u_StepDiv` geteilten Blockgröße. Mit dem Attribut `u_StepDivLog` können wir die Division durch eine schnellere Shiftoperation ersetzen. Die Schrittweite ist nun die zu `o_shiftedSize` nächstgrößere Zweierpotenz. Der Anfangswert dieser Zwischenrechnung ist `o_MinSize`. Damit stellen wir sicher, daß die Schrittweite nicht kleiner als die Minimalgröße werden kann. Mit der Formel `(o_size & - o_stepWidth)` wird die Blockgröße auf das nächstkleinere Vielfache der Schrittweite abgerundet. Dazu addieren wir die Schrittweite und subtrahieren abschließend die `StoreInfoSize` des Standardstores. Dieser Algorithmus ist sehr schnell, erfordert aber, daß wir die Blockgröße am Anfang um eins vermindern.

```
ct_RndStore:: t_Size ct_RndStore:: Round (t_Size o_size)
  {
```

```
o_size += sizeof (t_Size) + co_Store. StoreInfoSize () - 1;
if (o_size < o_MinSize)
  return o_MinSize - co_Store. StoreInfoSize ();
else
  {
  register t_Size o_stepWidth = o_MinSize;
  register t_Size o_shiftedSize = o_size >> u_StepDivLog;
  while (o_stepWidth <= o_shiftedSize)
    o_stepWidth <<= 1;
  return (o_size & - o_stepWidth) + o_stepWidth -
    co_Store. StoreInfoSize ();
  }
}
```

Zur Veranschaulichung führen wir nun ein Rechenbeispiel durch. Schritt-Teiler und `sizeof (t_Size)` seien gleich vier. Der um eins vergrößerte Zweierlogarithmus von vier ist gleich drei. Die Blockgröße betrage 184 Bytes. Am Anfang zählen wir 4 + 4 - 1 dazu und erhalten 191. Die Hilfsvariable `o_shiftedSize` erhält den Wert 191 >> 3 gleich 23. In der `while`-Schleife wird die nächstgrößere Zweierpotenz berechnet. Wir erhalten die Schrittweite 32. Mit (191 & - 32) wird die Blockgröße auf 160 abgerundet. Wir addieren 32 und subtrahieren die `StoreInfoSize`. Das Ergebnis lautet 188.

Ein Block dieser Größe wird vom Standardstore angefordert. Die interne Größe (einschließlich `StoreInfoSize`) beträgt 192 Bytes und ist, wie erhofft, eine runde Zahl (6 * 32). Der Roundstore nutzt die ersten vier der 188 Bytes für die Speicherung der Blockgröße. Nutzbar bleiben 184 Bytes. Mit dem Anfangswert 185 liefert die Methode `Round` den Wert 220. Davon sind 216 Bytes nutzbar.

Die dynamischen Storeklassen `ct_StdStore` und `ct_RndStore` bilden die Grundlage unserer Speicherverwaltung. Alle anderen Stores, die wir kennen lernen werden, bauen darauf auf. Abbildung 3-2 faßt die Attribute und Methoden beider Klassen zusammen. Ein statisches Attribut ist kein echter Teil eines Objekts. Es gehört allen Instanzen gemeinsam. Deshalb erscheint die Beziehung zwischen Standard- und Roundstore als Objekt-Verbindung (durchgehende Linie) und nicht als Teil-Ganzes-Beziehung (Dreieck).

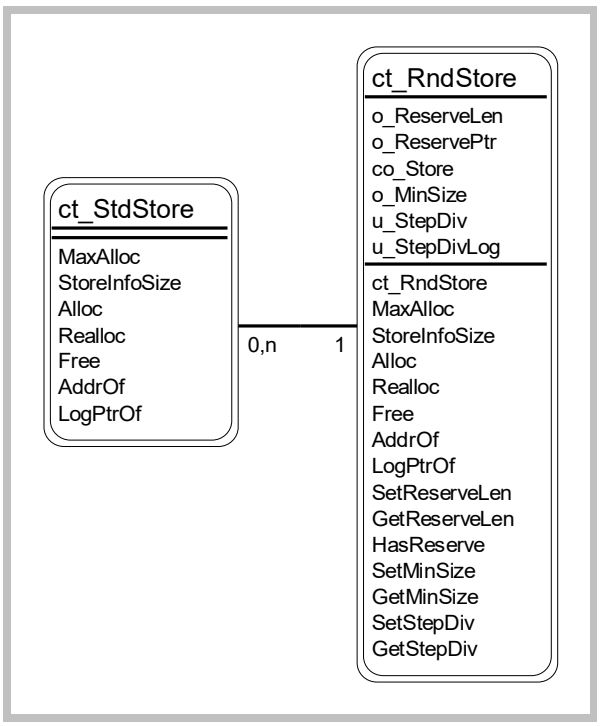

Abb. 3-2: *Standard- und Roundstore*

3.1.2 Globale Stores

Logische Zeiger und Stores sind eine Verallgemeinerung vorhandener Programmiertechniken. Eine feste Speicherverwaltung wird durch ein C++-Array realisiert. Indizes sind nur *lokal* im zugehörigen Array gültig. Für dynamische Speicheranforderungen nutzten wir bisher die C-Standardbibliothek. C++-Zeiger, die durch `malloc` bereitgestellt wurden, besitzen eine *globale* Gültigkeit. Analog dazu sind Stores mit einer festen Speicherverwaltung meist Bestandteil eines einzelnen Objekts, zum Beispiel eines Containers. Sie besitzen nur einen lokalen Gültigkeitsbereich. Dynamische Storeklassen werden hingegen als globale Objekte verwendet.

Wollen wir zum Beispiel eine spezialisierte dynamische Speicherverwaltung für eine Stringklasse einsetzen, sollte nicht jedes Stringobjekt über einen eigenen Store verfügen. Sinnvoller ist die Definition eines globalen Storeobjekts, das alle Instanzen der Stringklasse gemeinsam nutzen. Ein Stringobjekt kann auf zwei Arten mit dem globalen Store verbunden werden. Entweder enthält jeder String einen Zeiger auf den Store, oder die Stringklasse ist ein generischer Datentyp (ein Template) und besitzt den Store als Parameter. Im ersten Fall müssen wir einigen Overhead inkauf nehmen. Jedes Stringobjekt wird um einen C++-Zeiger vergrößert. Zum Kombinieren der Stringklasse mit verschiedenen Storeklassen müßte sie auf eine abstrakte Storebasisklasse mit virtuellen Methoden verweisen.

Die Verwendung einer Storeklasse als Templateparameter führt zu einer besseren Performance. Dabei stehen wir jedoch vor einem technischen Problem. Als Parameter für C++-Templates sind weder Methoden noch globale Objekte zugelassen. Möglich sind nur Datentypen und primitive Konstanten (Zahlen und Adressen). Ein globales Storeobjekt kann nicht als Parameter dienen, sondern nur seine Adresse. Der direkte Zugriff auf ein Objekt mit einem Zeiger widerspricht jedoch dem Konzept der Datenkapselung. In dynamisch gelinkten Bibliotheken kann es sogar zu einem Fehlverhalten des Programms führen. Auf globale Objekte und statische Attribute der Klassen sollte stets mit Nicht-Inline-Methoden zugegriffen werden.

C++ ist eine umfangreiche Programmiersprache. Sie enthält hardwarenahe Techniken (zum Beispiel die Zeigerarithmetik) und komfortable Konzepte (zum Beispiel virtuelle Basisklassen). Beim Parametrisieren eines Klassentemplates mit einem globalen Objekt läßt sie uns jedoch im Strich. Zum Glück verfügt jeder C++-Compiler über einen Präprozessor. Durch diese Hintertür können wir das Problem elegant lösen. Präprozessormakros enthalten einfache Textoperationen, die vor dem eigentlichen C++-Compiler ausgeführt werden. Der ##-Operator ist sogar die einzige Möglichkeit zur Bildung neuer Namen.

Benötigt wird eine Methodenschnittstelle für den indirekten Zugriff auf ein globales Objekt. Wir wollen die Schnittstelle als Templateparameter nutzen und verpacken sie in einer Klasse. Dieser geben wir das allgemeine Store-Interface, denn sie mappt die Funktionalität eines Storeobjekts. Die Klasse besitzt keine Attribute. Dennoch ist ihre Größe ungleich Null. Der C++-Standard besagt, daß jedes Objekt eine eindeutige Adresse besitzen muß. Einer Klasse ohne Attribute und virtuelle Tabellenzeiger weist der Compiler die Größe ein Byte zu. Da die Methoden auf keine Attribute zugreifen, können wir sie `static` deklarieren. Der Aufruf einer statischen Methode ist geringfügig schneller, denn sie enthält keinen `this`-Zeiger. Zum Erzeugen der Klasse definieren wir zwei Makros. In `GLOBAL_STORE_DCL` wird die Klasse deklariert. `GLOBAL_STORE_DEF` enthält die Definition ihrer Methoden.

```
#define GLOBAL_STORE_DCL(t_store, Obj, Size, t_size)              \
  class ct_ ## Obj ## Size ## Store                               \
    {                                                             \
  public:                                                         \
```

```
    typedef t_size             t_Size;                              \
    typedef t_store:: t_Pointer t_Pointer;                          \
    static unsigned long MaxAlloc ();                               \
    static unsigned    StoreInfoSize ();                            \
    static t_Pointer   Alloc (t_Size);                              \
    static t_Pointer   Realloc (t_Pointer, t_Size);                 \
    static void        Free (t_Pointer);                            \
    void *             AddrOf (t_Pointer o_p) const { return o_p; } \
    t_Pointer          LogPtrOf (void * pv) const { return pv; }    \
    static t_store *   GetStore ();                                 \
    };

#define GLOBAL_STORE_DEF(t_store, Obj, Size)                        \
  unsigned long ct_ ## Obj ## Size ## Store:: MaxAlloc ()           \
    { return co_ ## Obj ## Store. MaxAlloc (); }                    \
  unsigned ct_ ## Obj ## Size ## Store:: StoreInfoSize ()           \
    { return co_ ## Obj ## Store. StoreInfoSize (); }               \
  ct_ ## Obj ## Size ## Store:: t_Pointer                           \
  ct_ ## Obj ## Size ## Store:: Alloc (t_Size o_s)                  \
    { return co_ ## Obj ## Store. Alloc (o_s); }                    \
  ct_ ## Obj ## Size ## Store:: t_Pointer                           \
  ct_ ## Obj ## Size ## Store:: Realloc (t_Pointer o_p, t_Size o_s) \
    { return co_ ## Obj ## Store. Realloc (o_p, o_s); }             \
  void ct_ ## Obj ## Size ## Store:: Free (t_Pointer o_p)           \
    { co_ ## Obj ## Store. Free (o_p); }                            \
  t_store * ct_ ## Obj ## Size ## Store:: GetStore ()               \
    { return & co_ ## Obj ## Store; }
```

Der Parameter `t_store` bezeichnet die Storeklasse, zum Beispiel `ct_StdStore` oder `ct_RndStore`. `Obj` enthält eine Identität für das globale Objekt. Sie muß bei jedem Objekt anders gewählt werden. Zum Beispiel wird aus der Identität `String` das globale Objekt `co_StringStore` generiert. Zu einem globalen Store können mehrere Klassen erzeugt werden. Diese unterscheiden sich durch ihren Namen und den geschachtelten Datentyp `t_Size`. Der Makroparameter `Size` enthält eine Kurzbezeichnung zur Namensbildung, `t_size` den entsprechenden C++-Typ. `Size` und `t_size` müssen inhaltlich zusammenpassen, zum Beispiel `Char` und `unsigned char`.

Die Methode `AddrOf` eines Stores ist zeitkritisch. Sie wird bei jedem Zugriff auf einen Speicherblock aufgerufen. In den Klassen `ct_StdStore` und `ct_RndStore` ist ein logischer Zeiger gleich der Adresse des Blocks. Die daraus generierten Klassen nutzen diese Eigenschaft und definieren die Methoden `AddrOf` und `LogPtrOf inline`. Verwenden wir die Makros `GLOBAL_STORE_DCL` und `GLOBAL_STORE_DEF` für andere Storeklassen, müssen wir auf die Bedingung `o_ptr == AddrOf (o_ptr)` achten.

Eine generierte Storeklasse enthält nur das normale Interface. Für Erweiterungen wie die Minimalgröße benötigen wir das globale Objekt. Dazu dient die Methode `GetStore`. Zum Beispiel wird im Makro `GLOBAL_STORE_DCL (ct_RndStore, String, Int, unsigned int)` die Klasse `ct_StringIntStore` deklariert. Verwenden wir ein Objekt dieser Klasse (zum Beispiel `co_store`), können wir die Methode `SetMinSize` nicht direkt aufrufen. `GetStore` liefert aber den Verweis auf das globale Objekt von Typ `ct_RndStore`. Die Minimalgröße können wir mit dem Aufruf `co_store. GetStore ()-> SetMinSize (32)` ändern. Das expandierte Makro zur Deklaration der Klasse `ct_StringIntStore` enthält den folgenden Text.

```
class ct_StringIntStore
  {
public:
  typedef unsigned int            t_Size;
  typedef ct_RndStore:: t_Pointer t_Pointer;
  static unsigned long MaxAlloc ();
  static unsigned      StoreInfoSize ();
  static t_Pointer     Alloc (t_Size);
```

```
  static t_Pointer     Realloc (t_Pointer, t_Size);
  static void          Free (t_Pointer);
  void *               AddrOf (t_Pointer o_p) const { return o_p; }
  t_Pointer            LogPtrOf (void * pv) const { return pv; }
  static ct_RndStore * GetStore ();
  };
```

Zur Definition der Methoden einer generierten Klasse benötigen wir den Makroparameter `t_size` nicht. Die Kurzbezeichnung `Size` ist jedoch zur Namensbildung weiterhin erforderlich. Im Makro `GLOBAL_STORE_DEF (ct_StdStore, Std, Char)` werden die Methoden der Klasse `ct_StdCharStore` definiert. Sie greifen auf das globale Objekt `co_StdStore` zu.

```
unsigned long ct_StdCharStore:: MaxAlloc ()
  { return co_StdStore. MaxAlloc (); }
unsigned ct_StdCharStore:: StoreInfoSize ()
  { return co_StdStore. StoreInfoSize (); }
ct_StdCharStore:: t_Pointer
ct_StdCharStore:: Alloc (t_Size o_s)
  { return co_StdStore. Alloc (o_s); }
....
ct_StdStore * ct_StdCharStore:: GetStore ()
  { return & co_StdStore; }
```

Als Datentyp `t_Size` einer Storeklasse können die vorzeichenlosen Versionen von `int`, `char`, `short` und `long` eingesetzt werden. Zum Generieren der entsprechenden Klassen eines globalen Stores verwenden wir wieder zwei Makros. In `GLOBAL_STORE_DCLS` werden die vier Klassen und eine globale Funktion deklariert, mit der wir auf das Objekt zugreifen können. `GLOBAL_STORE_DEFS` enthält die Definitionen des globales Objekts und der Methoden der generierten Klassen.

```
#define GLOBAL_STORE_DCLS(t_store, Obj)                 \
  t_store * Get ## Obj ## Store ();                     \
  GLOBAL_STORE_DCL (t_store, Obj, Int,   unsigned int)  \
  GLOBAL_STORE_DCL (t_store, Obj, Char,  unsigned char) \
  GLOBAL_STORE_DCL (t_store, Obj, Short, unsigned short) \
  GLOBAL_STORE_DCL (t_store, Obj, Long,  unsigned long)

#define GLOBAL_STORE_DEFS(t_store, Obj)                 \
  t_store co_ ## Obj ## Store;                          \
  t_store * Get ## Obj ## Store ()                      \
    { return & co_ ## Obj ## Store; }                   \
  GLOBAL_STORE_DEF (t_store, Obj, Int)                  \
  GLOBAL_STORE_DEF (t_store, Obj, Char)                 \
  GLOBAL_STORE_DEF (t_store, Obj, Short)                \
  GLOBAL_STORE_DEF (t_store, Obj, Long)
```

Zur Erleichterung der Anwendung generieren wir für die dynamischen Storeklassen `ct_StdStore` und `ct_RndStore` je ein globales Objekt der Identität `Std` bzw. `Rnd`. Das Generieren weiterer Objekte, zum Beispiel für eine spezialisierte Stringklasse, bleibt dem Anwender überlassen. Die Deklaration der Klasse `ct_StdStore` ergänzen wir um das Makro `GLOBAL_STORE_DCLS (ct_StdStore, Std)`. Darin wird der folgende Text erzeugt.

```
ct_StdStore * GetStdStore ();    // Zugriffsfunktion auf globales Objekt
class ct_StdIntStore   { .... }; // Globale Klasse mit t_Size == u. int
class ct_StdCharStore  { .... }; // Globale Klasse mit t_Size == u. char
class ct_StdShortStore { .... }; // Globale Klasse mit t_Size == u. short
class ct_StdLongStore  { .... }; // Globale Klasse mit t_Size == u. long
```

Analog verfahren wir mit dem Roundstore. In der Headerdatei plazieren wir das Deklarationsmakro `GLOBAL_STORE_DCLS (ct_RndStore, Rnd)`. Das folgende Programmfragment

zeigt das expandierte Definitionsmakro `GLOBAL_STORE_DEFS (ct_RndStore, Rnd)`, das sich in der Implementierungsdatei befindet.

```
ct_RndStore co_RndStore;
ct_RndStore * GetRndStore ()
  { return & co_RndStore; }
unsigned long ct_RndIntStore:: MaxAlloc ()
  { return co_RndStore. MaxAlloc (); }
....
unsigned long ct_RndCharStore:: MaxAlloc ()
  { return co_RndStore. MaxAlloc (); }
....
unsigned long ct_RndShortStore:: MaxAlloc ()
  { return co_RndStore. MaxAlloc (); }
....
unsigned long ct_RndLongStore:: MaxAlloc ()
  { return co_RndStore. MaxAlloc (); }
....
```

Wir können die generierten Klassen leicht von der eigentlichen Storeklasse `ct_RndStore` unterscheiden. Sie enthalten im Namen die Kurzbezeichnung ihres Größentyps, zum Beispiel `ct_RndShortStore`. In der Anwendung dürfen wir sie nicht miteinander verwechseln. Die Klasse `ct_RndStore` besitzt eigene Attribute, zum Beispiel `o_MinSize`. Die generierten Klassen besitzen keine Attribute. Sie mappen die Funktionalität eines globalen Objekts und verfügen nur über das allgemeine Store-Interface. Diesen Unterschied können wir in einem Designdiagramm gut erkennen (siehe Abbildung 3-3).

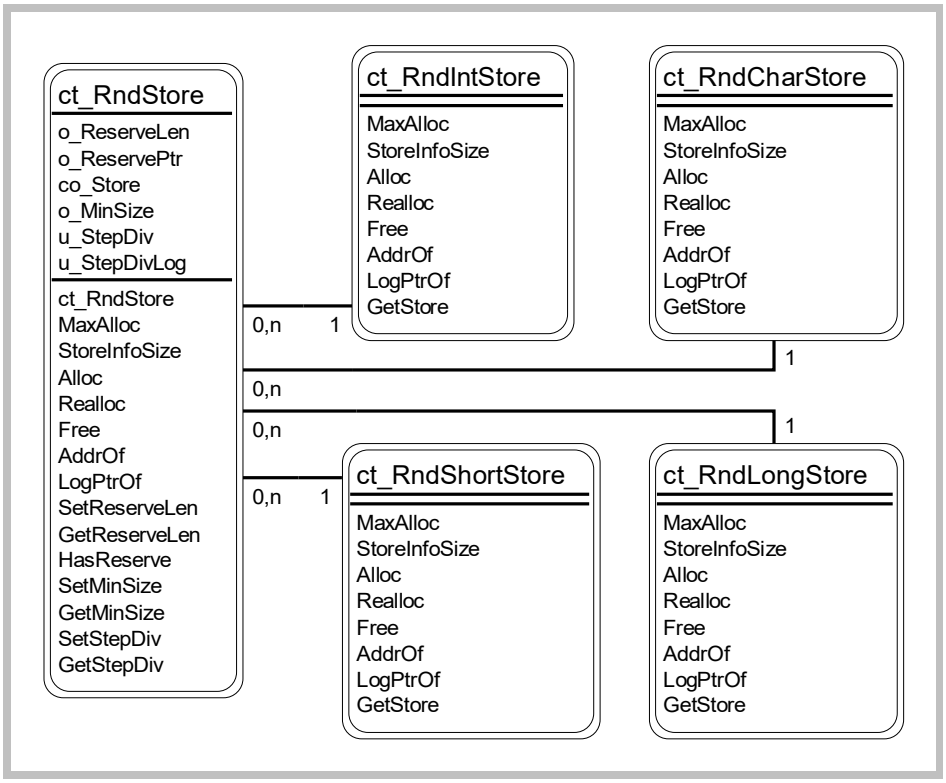

Abb. 3-3: *Globale Klassen des Roundstores*

Im Designdiagramm besitzen die globalen Klassen eine Objekt-Verbindung zur Klasse `ct_RndStore`. Die genaue Art der Verbindung können wir dem Diagramm jedoch nicht entnehmen. In Abbildung 3-4 wird eine andere Darstellungsform gewählt. Dort sehen wir, daß die Verbindung der Klassen über ein globales Objekt erfolgt.

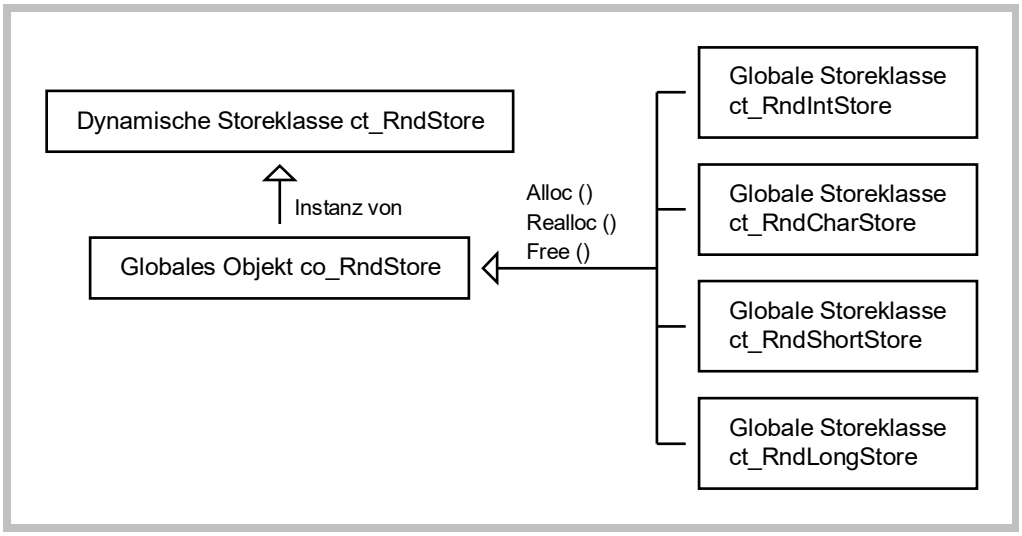

Abb. 3-4: *Andere Darstellung der Klassen*

3.1.3 Globale C++-Speicherverwaltung

Wollen wir einen Store für die globale C++-Speicherverwaltung einsetzen, müssen wir die vorgegebenen Operatoren `new` und `delete` neu definieren. Der Store muß den folgenden beiden Anforderungen genügen: Der Typ `t_Size` ist als `unsigned int` oder `unsigned long` definiert. Eine Adresse behält solange ihre Gültigkeit, bis sie mit `Realloc` oder `Free` geändert wird. Die Klassen `ct_StdStore` und `ct_RndStore` erfüllen diese Bedingungen. Beim Einsatz eines Stores für die globalen Operatoren `new` und `delete` müssen wird darauf achten, alle mit `new` angeforderten Blöcke mit `delete` und nicht mit der Standardfunktion `free` freizugegeben. Umgekehrt müssen wir alle mit `malloc` bereitgestellten Zeiger mit `free` an die Speicherverwaltung der Standardbibliothek zurückgeben.

Der folgende Programmausschnitt enthält die für eine eigene globale Speicherverwaltung nötigen Definitionen. Die Umrechnung von logischen Zeigern in Adressen und umgekehrt kann bei Standard- und Roundstores entfallen, denn für sie gilt `o_ptr == AddrOf (o_ptr)`. Für einen korrekten Programmierstil sollten die Zeiger dennoch umgerechnet werden. Die Methoden `AddrOf` und `LogPtrOf` sind `inline` definiert. Sie geben die Zeiger unverändert weiter und belasten nicht die Rechenzeit.

```
#include <stddef.h> // Für globalen Typ size_t

void * operator new (size_t u_size)
  {
  return GetRndStore ()-> AddrOf (GetRndStore ()-> Alloc (u_size));
  }

void * operator new [] (size_t u_size)
  {
  return GetRndStore ()-> AddrOf (GetRndStore ()-> Alloc (u_size));
  }

void operator delete (void * pv)
  {
  GetRndStore ()-> Free (GetRndStore ()-> LogPtrOf (pv));
  }

void operator delete [] (void * pv)
  {
  GetRndStore ()-> Free (GetRndStore ()-> LogPtrOf (pv));
```

```
  }

int main ()
  {
  char * pc1 = new char [30];
  char * pc2 = new char [25];
  ....
  delete [] pc1; // OK
  free (pc2);    // Crash !!
  }
```

3.1.4 Dynamischer Speicherblock

In vielen Klassen eines Programms, zum Beispiel Strings und Arraycontainern, wird ein Speicherblock dynamischer Länge benötigt. Der Speicher wird von einem Store angefordert. Für den späteren Zugriff muß ein Verweis in einem Objekt gehalten werden. Dieses Zugriffsobjekt nennen wir im folgenden kurz **Block**. Es enthält einen Zeiger auf den Speicherbereich und dessen Länge (siehe Abbildung 3-5).

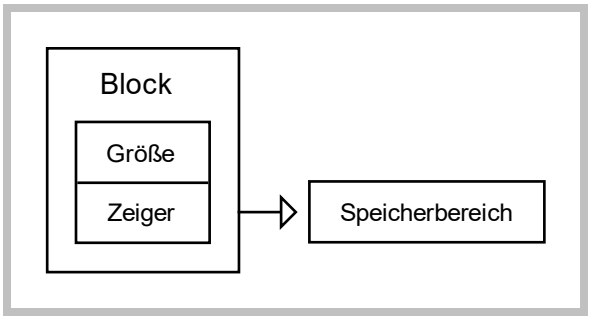

Abb. 3-5: *Blockkonzept*

Der Block stellt keine umfangreiche Funktionalität zur Verfügung, ist aber ein allgemeines, wiederverwendbares Konzept. Analog zu den Stores und Containern definieren wir dafür keine abstrakte Basisklasse, sonden nur ein Interface. Die Größe des dynamischen Speicherbereichs muß abgefragt und neu gesetzt werden können. Für den Zugriff auf die darin enthaltenen Daten wird seine Adresse benötigt. Sie kann als `void *` oder `char *` abgefragt werden. Beide Formen werden häufig verlangt. Der Datentyp für die Größe kann sich von Block zu Block unterscheiden. Er ist als geschachtelter Typ in der Blockklasse enthalten.

```
class ct_AnyBlock
  {
public:
  typedef unsigned int t_Size;

  t_Size              GetSize () const;
  void                SetSize (t_Size o_newSize);
  void *              GetAddr () const;
  char *              GetCharAddr () const;
  };
```

Häufig werden Blockklassen benötigt, die ihren Speicher von einem gobalen Store anfordern. Dafür definieren wir ein Klassentemplate. Es ist eine erste Anwendung der generierten Storeklassen. Über das normale Blockinterface hinaus besitzt es weitere Methoden. Zum Beispiel erhalten wir mit `GetPtr` den logischen Zeiger des Speicherbereichs. Die Methoden des Blocktemplates sind so einfach, daß alle `inline` definiert werden können. Im folgenden

Programmausschnitt sehen wir die Deklaration des Klassentemplates und die Implementierung dreier Methoden.

```
template <class t_store>
  class gct_Block
    {
  public:
    typedef t_store:: t_Size    t_Size;
    typedef t_store:: t_Pointer t_Pointer;
  protected:
    t_Size               o_Size;
    t_Pointer            o_Ptr;
    static t_store       o_Store;
  public:
    inline               gct_Block ();
    inline               gct_Block (const gct_Block & co_init);
    inline               ~gct_Block ();
    inline gct_Block &   operator = (const gct_Block & co_asgn);
    inline t_Size        GetSize () const;
    inline void          SetSize (t_Size o_newSize);
    inline t_Pointer     GetPtr () const;
    inline void *        GetAddr () const;
    inline char *        GetCharAddr () const;
    inline t_store *     GetStore () const;
    };

template <class t_store>
  inline void gct_Block <t_store>:: SetSize (t_Size o_newSize)
    {
    o_Size = o_newSize;
    o_Ptr = o_Store. Realloc (o_Ptr, o_Size);
    }

template <class t_store>
  inline gct_Block <t_store>:: t_Pointer
  gct_Block <t_store>:: GetPtr () const
    {
    return o_Ptr;
    }

template <class t_store>
  inline void * gct_Block <t_store>:: GetAddr () const
    {
    return o_Store. AddrOf (o_Ptr);
    }
```

Vom Templateparameter `t_store` wird das Attribut `o_Store` gebildet. Es stellt die Funktionalität eines globalen Storeobjekts zur Verfügung und ist für alle Instanzen der Blockklasse gleich. Da die Größe eines C++-Objekts ungleich Null sein muß, würde `o_Store` das Blockobjekt um ein Dummy-Byte vergrößern. Deshalb deklarieren wir es `static`. Die Verwendung statischer Attribute in `inline`-Methoden (siehe `SetSize` und `GetAddr`) kann in dynamisch gelinkten Bibliotheken zu einem Fehlverhalten des Programms führen. Bei einer Instanz einer globalen Storeklasse besteht diese Gefahr nicht, denn das Objekt enthält nur ein Dummy-Byte und statische Methoden.

Der Konstruktor des Blocks initialisiert die Attribute `o_Size` und `o_Ptr` auf Null. Der Destruktor gibt den angeforderten Speicher an den Store zurück. Kopier-Konstruktor und Gleich-Operator übernehmen nur die Größe des zu kopierenden Blocks, nicht den Inhalt des Speicherbereichs. Die Kopiersemantik kann sehr unterschiedlich sein und muß vom Blockanwender implementiert werden. Ein String kopiert die Zeichenkette binär. In einem Arraycontainer müssen die Objekte einzeln kopiert werden.

```
template <class t_store>
  inline gct_Block <t_store>:: gct_Block (const gct_Block & co_init)
    {
    o_Size = co_init. o_Size;
    o_Ptr = o_Store. Alloc (co_init. o_Size);
    }

template <class t_store>
  inline gct_Block <t_store> &
  gct_Block <t_store>:: operator = (const gct_Block & co_asgn)
    {
    SetSize (co_asgn. o_Size);
    return * this;
    }
```

Ein C++-Template besitzt Ähnlichkeit mit einem Präprozessormakro. Ersetzen wir im Templatetext die formalen durch die aktuellen Parameter, erhalten wir eine normale Klasse. Werden zusätzlich die geschachtelten Typen der Storeklasse aufgelöst, ergibt sich für die Blockklasse `gct_Block <ct_RndCharStore>` die folgende Deklaration.

```
class gct_Block <ct_RndCharStore>
  {
public:
  typedef unsigned char  t_Size;
  typedef void *         t_Pointer;
protected:
  t_Size                 o_Size; // Ein Byte
  t_Pointer              o_Ptr;  // Vier Bytes
  static ct_RndCharStore o_Store;
public:
  ....
  inline ct_RndCharStore * GetStore () const; // Liefert Adr. von o_Store
  };
```

Wir können Instanzen dieser Blockklasse bilden, zum Beispiel `co_block`. Das Objekt umfaßt fünf Bytes. Die Größe des Speicherbereichs ist durch den Datentyp `unsigned char` begrenzt und kann maximal 255 Bytes betragen. Im Normalfall greifen wir mit der Funktion `GetRndStore` auf den zugehörigen globalen Store `co_RndStore` zu. Verwenden wir mehrere globale Roundstores, ist nicht immer der Store bekannt, von dem ein Block seinen Speicher anfordert. Wir erreichen ihn auch über den Block. Die Blockmethode `GetStore` liefert die Adresse des Attributs `o_Store`. Mit der Methode `GetStore` dieses Attributs erhalten wir einen Zeiger auf das globale Storeobjekt. Zum Verändern der Minimalgröße des zugehörigen Roundstores ist die folgende Anweisung erforderlich.

```
co_block. GetStore ()-> GetStore ()-> SetMinSize (32);
```

3.1.5 Eine Blockanwendung - String

Als erste Anwendung des Blockkonzepts lernen wir ein Stringtemplate kennen. Es wird mit einer Blockklasse parametrisiert und verwendet nur das normale Blockinterface. Eine Klasse mit der erweiterten Funktionalität des Templates `gct_Block` kann auch als Parameter übergeben werden. Die Stringklasse besitzt keine eigenen Attribute. Auf Länge und Speicherbereich greift sie über das Blockinterface zu. Ebenso benötigt sie keinen Destruktor. Die Basisklasse gibt am Ende den belegten Speicher frei. Die Methodenschnittstelle übernehmen wir von der Stringklasse unseres Beispielprogramms OHelp. Für Längenangaben wird der lokale Typ `t_Size` des Blocks verwendet. Der folgende Programmausschnitt enthält die Deklaration des Klassentemplates und die Implementierung zweier Methoden.

```
template <class t_block>
```

```
  class gct_String: private t_block
    {
  protected:
    inline void          SetLen (t_Size o_len);
  public:
    inline               gct_String ();
    inline               gct_String (const char * pc_init);
    inline               gct_String (const gct_String & co_init);
    gct_String &         operator = (const char * pc_asgn);
    gct_String &         operator = (const gct_String & co_asgn);
    inline t_Size        GetLen () const;
    inline const char *  GetStr (t_Size o_pos = 0) const;
    void                 Insert (t_Size o_pos, const char * pc_ins);
    void                 Delete (t_Size o_pos, t_Size o_len);
    inline char &        operator [] (t_Size o_pos) const;
    inline const t_block * GetConstBlock () const;
    };

template <class t_block>
  inline void gct_String <t_block>:: SetLen (t_Size o_len)
    {
    SetSize (o_len + 1);
    }

template <class t_block>
  void gct_String <t_block>:: Insert (t_Size o_pos, const char * pc_ins)
    {
    ASSERT (o_pos <= GetLen ());
    ASSERT (pc_ins != 0);
    t_Size o_inslen = (t_Size) strlen (pc_ins);
    if (o_inslen > 0)
      {
      SetLen (GetLen () + o_inslen);
      memmove (GetCharAddr () + o_pos + o_inslen, GetCharAddr () + o_pos,
        GetLen () - o_pos - o_inslen + 1);
      memcpy  (GetCharAddr () + o_pos, pc_ins, o_inslen);
      }
    }
```

Solange die Bedingung `GetLen () == strlen (GetStr ())` gilt, befindet sich ein Stringobjekt in einem konsistenten Zustand. Um die Konsistenz zu gewährleisten, werden verändernde Blockmethoden, zum Beispiel `SetSize`, nur innerhalb der Stringmethoden aufgerufen. Einem Stringanwender dürfen sie nicht zugänglich sein. Deshalb erbt die Stringklasse privat von der übergebenen Blockklasse. Mit der Methode `GetConstBlock` werden dem Anwender des Strings eventuelle Erweiterungen des Blocks zugänglich gemacht. Sie liefert einen Zeiger auf die konstante Basisklasse.

```
template <class t_block>
  inline const t_block * gct_String <t_block>:: GetConstBlock () const
    {
    return this;
    }
```

Eine bessere Lösung für dieses Problem wäre eine konstante Vererbung. Dabei könnte ein Anwender der abgeleiteten Klasse nur konstante Methoden der Basisklasse aufrufen. Die Programmiersprache C++ bietet dafür aber keine Möglichkeit.

```
template <class t_block>
  class gct_String: const public t_block // Hilfreich, aber kein C++
    { ....
```

Ebenso zahlreich wie die Blockklassen sind die möglichen Stringklassen. Zum Beispiel fordert die Templateklasse gct_String <gct_Block <ct_RndShortStore> > ihren Speicher vom globalen Objekt co_RndStore an. Ein Stringobjekt benötigt insgesamt sechs Bytes, vier Bytes für den Zeiger (void *) und zwei Bytes für die Länge (unsigned short). Die Zeichenkette kann maximal 65535 Bytes enthalten. Wir können den Speicher auch aus einem Roundstore anfordern, der andere Werte für Minimalgröße und Schritt-Teiler als das vordefinierte Objekt co_RndStore besitzt. Das folgende Programmfragment demonstriert die erforderlichen Schritte.

```
// In einer Headerdatei plazieren
GLOBAL_STORE_DCLS (ct_RndStore, String)
typedef gct_Block <ct_StringIntStore> t_StringBlock;
typedef gct_String <t_StringBlock> t_String;

// In einer Implementierungsdatei plazieren
GLOBAL_STORE_DEFS (ct_RndStore, String)

int main ()
  {
  GetStringStore ()-> SetMinSize (32);
  GetStringStore ()-> SetStepDiv (2);
  t_String o_string;
  ....
  }
```

Die Makros generieren das globale Objekt co_StringStore und die zugehörigen Storeklassen. Mit ct_StringIntStore parametrisieren wir eine Blockklasse und nennen sie t_StringBlock. Sie dient als Parameter des Stringtemplates. Zur Abkürzung wird auch die Stringklasse per Typdefinition in t_String umbenannt. Am Programmbeginn stellen wir Minimalgröße und Schritt-Teiler des globalen Objekts co_StringStore ein. Danach können wir Instanzen der Stringklasse bilden, zum Beispiel o_string. In Abbildung 3-6 sehen wir die wichtigsten Klassen, von denen t_String abhängt.

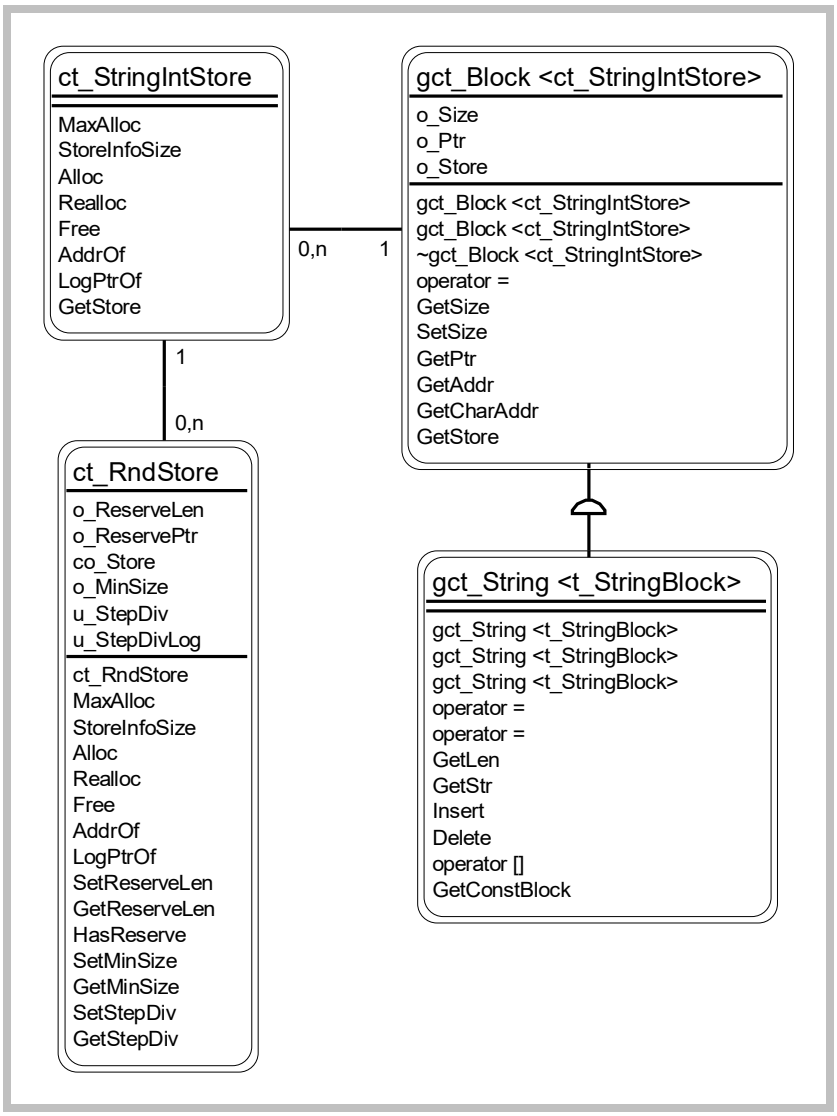

Abb. 3-6: *An t_String beteiligte Klassen*

Der dynamische Speicherblock eines Strings umfaßt mindestens ein Zeichen, das abschließende Nullzeichen. Es wird bei der Länge nicht mitgerechnet, ist aber im zugrundeliegenden Block enthalten. Es wird schon im Konstruktor initialisiert.

```
template <class t_block>
  inline gct_String <t_block>:: gct_String ()
    {
    SetLen (0);
    * GetCharAddr () = '\0';
    }
```

In Tabelle 3-1 sehen wird die Methodenaufrufe zum Anfordern dieses einen Bytes. Der Konstruktor (Schritt 1) ruft die Methode `SetLen` auf (Schritt 2). Sie erhöht den übergebenen Wert um eins und gibt ihn an die Blockmethode `SetSize` weiter (Schritt 3). Die Blockklasse verwendet ihr statisches Attribut `o_Store` zum Verändern der Größe (Schritt 4). Die globale Storeklasse gibt den Methodenaufruf an das zugehörige Objekt `co_StringStore` weiter (Schritt 5). Da der übergebene Zeiger gleich Null ist, ruft `Realloc` die Methode `Alloc` auf (Schritt 6). Diese rundet die Größe (Schritt 7) und fordert vom statischen Attribut `co_Store` den Speicher an (Schritt 8). Der Standardstore mappt die Anforderung auf die Standardfunktion `malloc` (Schritt 9).

Schritt	Objekt	Methode
1	o_string	t_String:: t_String ()
2	o_string	t_String:: SetLen (0)
3	o_string	t_StringBlock:: SetSize (1)
4	t_StringBlock:: o_Store	ct_StringIntStore:: Realloc (0, 1)
5	co_StringStore	ct_RndStore:: Realloc (0, 1)
6	co_StringStore	ct_RndStore:: Alloc (1)
7	co_StringStore	ct_RndStore:: Round (1)
8	ct_RndStore:: co_Store	ct_StdStore:: Alloc (28)
9		malloc (28)

Tab. 3-1: *Speicheranforderung im Stringkonstruktor*

Nachdem der Speicher angefordert wurde, muß das erste Byte mit dem Wert Null initialisiert werden. Die für den Zugriff nötigen Schritte sehen wir in Tabelle 3-2. Der Konstruktor ruft die Blockmethode `GetCharAddr` auf. Diese ruft über das statische Attribut `o_Store` die Methode `ct_StringIntStore:: AddrOf (o_Ptr)` auf. Beide Methoden sind `inline` definiert und enthalten keine Berechnungen. Der Zugriff ist genauso schnell wie über einen C++-Zeiger.

Schritt	Objekt	Methode
1	o_string	t_String:: t_String ()
2	o_string	t_StringBlock:: GetCharAddr ()
3	t_StringBlock:: o_Store	ct_StringIntStore:: AddrOf (o_Ptr)

Tab. 3-2: *Speicherzugriff im Stringkonstruktor*

3.2 Speicher nach Maß

3.2.1 Fester Store im Block

Eine einfache feste Speicherverwaltung wird mit Hilfe eines Blocks realisiert. Wir nennen sie kurz **Blockstore**. Für eine gute Performance definieren wir auch dafür ein Klassentemplate. Es wird mit einer Blockklasse parametrisiert. Ein weiterer Parameter ist die Größe der Blöcke, die bereitgestellt werden sollen. Eine Speicherung der Größe als Attribut ergibt wenig Sinn, denn es ist eine Konstante. Sie müßte mit dem Konstruktor initialisiert werden und würde sich später nicht mehr ändern, aber Speicher belegen. Eine bessere Lösung ist die Übergabe als Templateparameter. Im folgenden Programmausschnitt sehen wir die Deklaration des Klassentemplates für einen Blockstore.

```
template <class t_block, unsigned u_fixSize>
  class gct_BlockStore: private t_block
    {
  public:
    typedef t_block:: t_Size t_Size;
    typedef t_block:: t_Size t_Pointer;
  private:
```

```
    t_Pointer            o_FirstFree;

    inline t_Pointer *   IdxAddrOf (t_Pointer o_ptr) const;
  public:
    inline               gct_BlockStore ();
    inline unsigned long MaxAlloc () const;
    inline unsigned      StoreInfoSize () const;
    t_Pointer            Alloc (t_Size o_size);
    t_Pointer            Realloc (t_Pointer o_ptr, t_Size o_size);
    void                 Free (t_Pointer o_ptr);
    inline void *        AddrOf (t_Pointer o_ptr) const;
    inline t_Pointer     LogPtrOf (void * pv_adr) const;
    inline t_Pointer     LastIdx () const;
    inline bool          HasFree () const;
    inline const t_block * GetConstBlock () const;
    };
```

Um dem Anwender eventuelle Erweiterungen des Blocks zugänglich zu machen, enthält auch der Blockstore eine Methode `GetConstBlock`. Der Größentyp `t_Size` wird vom Block übernommen. In einem Blockstore ist der Zeigertyp `t_Pointer` gleich dem Größentyp. Die verwalteten Speicherblöcke werden aufsteigend numeriert. Die fortlaufende Nummer innerhalb des umfassenden Blocks ergibt den logischen Zeiger. Die Zählung beginnt mit dem Wert Eins, denn der Zeigerwert Null ist per Definition ungültig.

Die Methode `MaxAlloc` liefert als Resultat den Templateparameter `u_fixSize`. Größere Speicherbereiche kann ein Blockstore nicht zur Verfügung stellen. Die `StoreInfoSize` beträgt null Bytes. Das verdeutlicht die hohe Speicherauslastung. `LastIdx` berechnet den größten gültigen Zeigerwert des Blockstores. `HasFree` gibt Auskunft darüber, ob Elemente in der Freiliste enthalten sind. Die Methode `AddrOf` liefert die zu einem logischen Zeiger (einem Index) gehörende Speicheradresse. Der um eins verminderte Index wird mit `u_fixSize` multipliziert. Wir erhalten die Byteposition innerhalb des umfassenden Blocks. Diese wird zur Anfangsadresse addiert. Der logische Nullzeiger wird gesondert behandelt.

```
template <class t_block, unsigned u_fixSize>
  inline gct_BlockStore <t_block, u_fixSize>:: t_Pointer
  gct_BlockStore <t_block, u_fixSize>:: LastIdx () const
    {
    return GetSize () / u_fixSize;
    }

template <class t_block, unsigned u_fixSize>
  inline void *
  gct_BlockStore <t_block, u_fixSize>:: AddrOf (t_Pointer o_ptr) const
    {
    if (o_ptr == 0)
      return 0;
    else
      {
      ASSERT (o_ptr <= LastIdx ());
      return GetCharAddr () + (unsigned) (o_ptr - 1) * u_fixSize;
      }
    }
```

In einem Blockstore verursacht die Implementierung der Freiliste den größten Aufwand. Der Freispeicher verwaltet sich wie in einem dynamischen Store selbst. Die freien Blöcke bilden eine einfach verkettete Liste. Jedes Element enthält den logischen Zeiger des Nachfolgers. Deshalb muß für einen Blockstore die Bedingung `u_fixSize >= sizeof (t_Pointer)` gelten. Eine unsortierte Freiliste ist einfach zu handhaben. Dabei ist aber die Wahrscheinlichkeit gering, daß am physischen Ende des umfassenden Blocks etwas frei wird und dieser verkleinert werden kann.

Die private Methode IdxAddrOf dient der Verwaltung der Freiliste. Sie wandelt den untypisierten C++-Zeiger der Methode AddrOf in einen typisierten um. Enthält der logische Zeiger o_ptr den Index eines Elements der Freiliste, erhalten wir mit * IdxAddrOf (o_ptr) den Index des Nachfolgers.

Unsere Implementierung des Blockstores verfügt über eine sortierte Freiliste. Das Attribut o_FirstFree verweist auf das Freielement mit dem kleinsten Index. Jedes Element der Freiliste enthält den Index des nächstgrößeren. Die Methode Alloc versucht, das erste Element aus der Freiliste zu entnehmen. Ist die Freiliste leer, wird der Block um u_fixSize Bytes vergrößert und der Index des letzten Elements zurückgegeben. Beim Vergrößern kann ein Überlauf eintreten. Ist zum Beispiel der Größentyp gleich unsigned char, u_fixSize gleich zehn und die Blockgröße gleich 250 Bytes, ergibt (unsigned char) 250 + 10 den Wert Vier.

```
template <class t_block, unsigned u_fixSize>
  gct_BlockStore <t_block, u_fixSize>:: t_Pointer
  gct_BlockStore <t_block, u_fixSize>:: Alloc (t_Size o_size)
    {
    ASSERT (o_size <= u_fixSize);
    if (o_size == 0)
      return 0;
    else
      if (o_FirstFree != 0)
        {
        t_Pointer o_ptr = o_FirstFree;
        o_FirstFree = * IdxAddrOf (o_FirstFree);
        return o_ptr;
        }
      else
        if (GetSize () + (t_Size) u_fixSize < u_fixSize)
          return 0; // Überlauf
        else
          {
          SetSize (GetSize () + u_fixSize);
          ASSERT (GetAddr () != 0);
          return LastIdx ();
          }
    }
```

Die Implementierung von Realloc wird auf Alloc und Free zurückgeführt. Die Definition der Methode Free ist sehr umfangreich und wird hier nicht abgebildet. Das freizugebende Element wird in die Liste einsortiert. Befindet es sich am physischen Ende des umfassenden Blocks, wird dieser verkleinert. Dabei muß geprüft werden, ob sich unmittelbar davor weitere freie Elemente befinden.

Unser Blockstoretemplate verwendet nur das normale Interface eines Blocks. Es kann mit beliebigen Blockklassen parametrisiert werden. Die Klasse gct_BlockStore <10, gct_Block <ct_StdCharStore> > fordert ihren Speicher vom globalen Objekt co_StdStore an. Ein Objekt dieser Blockstoreklasse umfaßt sechs Bytes. Der dynamische Block kann bis zu 255 Bytes enthalten. Das entspricht 25 internen Blöcken der Größe 10 Bytes. Beim Anfordern des 26. Elements tritt der Überlauf ein.

Bei der Anwendung eines Blockstores müssen wir beachten, daß die Adresse eines bereitgestellten Blocks nur solange gültig bleibt, wie sich am Store nichts ändert. Fordern wir mit Alloc einen neuen Block an, und die Freiliste ist leer, muß der umfassende Block vergrößert werden. Dabei kann die Schrittweite des darunterliegenden globalen Stores überschritten werden. Der umfassende Block wird mit seinem gesamten Inhalt an eine andere Stelle im Speicher kopiert. Die Adressen der inneren Blöcke ändern sich, ihre logischen Zeiger behalten jedoch ihre Gültigkeit.

3.2.2 Ein Anwendungsbeispiel

Homogene Container sind typische Anwendungen des Blockstores. Wir werden sie einige Abschnitte später kennenlernen. Ein Blockstore kann auch für andere Zwecke eingesetzt werden, zum Beispiel eine spezialisierte Stringklasse. Treten in einem Programm viele Zeichenketten auf, die nicht länger als acht Zeichen werden (einschließlich Nullzeichen), ist eine dynamische Speicherverwaltung ungeeignet. Mit unseren bisherigen Mitteln würden wir dafür die Klasse `gct_String <gct_Block <ct_StdCharStore> >` verwenden. Sie benötigt vier Bytes für den Zeiger (`void *`) und ein Byte für die Länge (`unsigned char`), also insgesamt fünf Bytes. Die Zeichenkette wird vom globalen Objekt `co_StdStore` angefordert und belegt 16 Bytes. Das ist die interne Minimalgröße der dynamischen Speicherverwaltung.

Eine bessere Speicherauslastung erzielen wir mit einem Blockstore. Zum Generieren des globalen Storeobjekts und der zugehörigen Klassen können wir die bekannten Makros nicht einsetzen. Logische Zeiger eines Blockstores sind Indizes, und es gilt `o_ptr != AddrOf (o_ptr)`. Wir verwenden leicht geänderte Makros mit dem Suffix `STATIC`. Sie unterscheiden sich von den bekannten dadurch, daß die Methoden `AddrOf` und `LogPtrOf` nicht `inline`, sondern `static` definiert werden.

```
typedef gct_Block <ct_RndShortStore> t_RndShortBlock;
typedef gct_BlockStore <t_RndShortBlock, 8> t_Fix8Store;
GLOBAL_STORE_DCLS_STATIC (t_Fix8Store, Fix8)
GLOBAL_STORE_DEFS_STATIC (t_Fix8Store, Fix8)
```

In diesen Makros wird das globale Objekt `co_Fix8Store` generiert. Es verwaltet Blöcke der festen Größe acht Bytes. Der umfassende Block wird vom globalen Objekt `co_RndStore` angefordert. Die Blockgröße ist durch den Datentyp `unsigned short` auf 65535 Bytes begrenzt. Der globale Blockstore kann also maximal 8191 Acht-Byte-Blöcke bereitstellen. Seine logischen Zeiger sind ebenfalls vom Typ `unsigned short`.

Zum globalen Store werden vier Klassen generiert. Sie unterscheiden sich durch ihren geschachtelten Typ `t_Size`. Sinnvoll ist nur der Größentyp `unsigned char`, denn es können nicht mehr als acht Bytes angefordert werden. Mit dieser globalen Storeklasse können wir einen Block und damit einen String parametrisieren.

```
typedef gct_Block <ct_Fix8CharStore> t_Block8;
typedef gct_String <t_Block8> t_String8;
....
t_String8 o_string8;
```

Ein Objekt der Klasse `t_String8` umfaßt nur drei Bytes. Der Zeiger ist ein Index des globalen Blockstores und benötigt zwei Bytes. Die Größenangabe des Strings erfordert ein weiteres Byte. Die eigentliche Zeichenkette belegt acht Bytes im Blockstore (siehe Abbildung 3-7). Damit haben wir den Gesamtspeicherbedarf gegenüber der oben erwähnten Stringklasse auf etwa die Hälfte reduziert. Für ein einzelnes Stringobjekt lohnt sich dieser Aufwand nicht. Enthält aber eine Klasse zum Beispiel zehn Stringattribute, belegen sie mit der Stringklasse `t_String8` nur 30 Bytes.

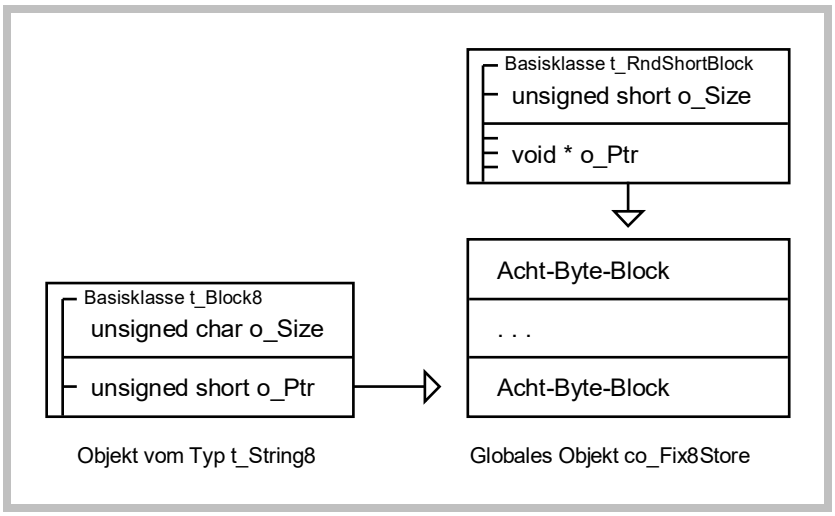

Abb. 3-7: *Speicherlayout der Objekte o_string8 und co_Fix8Store*

Die gute Speicherauslastung müssen wir mit einer Verlangsamung des Zugriffs bezahlen. Nachdem im Konstruktor des Objekts `o_string8` ein neuer Acht-Byte-Block bereitgestellt wurde, muß dem ersten Byte der Wert Null zugewiesen werden. In Tabelle 3-3 sehen wir die Teilschritte des Speicherzugriffs. Im Konstruktor (Schritt 1) wird die Blockmethode `GetCharAddr` aufgerufen (Schritt 2). Sie verwendet das statische Attribut `o_Store` (Schritt 3). Diesmal ist die Methode `AddrOf` der globalen Storeklasse nicht `inline` definiert. Sie gibt den Aufruf an das globale Objekt `co_Fix8Store` weiter (Schritt 4). Der globale Blockstore ruft die Methode `GetCharAddr` seiner Basisklasse `t_RndShortBlock` auf (Schritt 5). Die letzte Methode in dieser Folge ist `ct_RndShortStore:: AddrOf` (Schritt 6). Sie ist `inline` definiert und gibt den übergebenen Zeiger ohne Änderung zurück. Dieser Speicherzugriff sieht sehr aufwendig aus. Die meisten Methoden sind aber `inline` definiert. In der Summe ist er nicht viel langsamer als der Zugriff auf ein C++-Array mit einem Index.

Schritt	Objekt	Methode
1	o_string8	t_String8:: t_String8 ()
2	o_string8	t_Block8:: GetCharAddr ()
3	t_Block8:: o_Store	ct_Fix8CharStore:: AddrOf (o_Ptr)
4	co_Fix8Store	t_Fix8Store:: AddrOf (o_ptr)
5	co_Fix8Store	t_RndShortBlock:: GetCharAddr ()
6	t_RndShortBlock:: o_Store	ct_RndShortStore:: AddrOf (o_Ptr)

Tab. 3-3: *Speicherzugriff im Stringkonstruktor*

Einem neuen Objekt der Klasse `t_String8` wird im Konstruktor ein Block im globalen `co_Fix8Store` zugeordnet. Da die Größe des dynamischen Blocks nur zwischen ein und acht Bytes schwanken kann, behält das Objekt denselben logischen Zeiger bis zu seinem Destruktor. Dann wird der belegte Speicher freigegeben. Der globale Blockstore ändert sich also, wenn ein String erzeugt oder gelöscht wird. Während der Arbeit mit den Strings (`Insert`, `Delete` usw.) bleibt sein Aufbau konstant.

Die Methode `GetStr` liefert einen C++-Zeiger auf die Zeichenkette. Wir müssen beachten, daß dieser bei der Klasse `t_String8` nur solange gültig bleibt, wie am Blockstore nichts geändert wurde. Nach dem Erzeugen oder Löschen eines Strings sind alle vorher abgefragten C++-Zeiger nicht mehr gültig. Zum Beispiel dürfen wir den Konstruktor `gct_String (const char * pc)` nicht mit einem Rückgabewert von `GetStr` verwenden. Beim

Kopierkonstruktor tritt dieses Problem nicht auf, denn die Zeichenkette des zu kopierenden Strings wird erst abgefragt, nachdem der Block für das neue Objekt bereitgestellt wurde.

```
t_String8 o_string81 (o_string8);            // OK
t_String8 o_string82 (o_string8. GetStr ()); // Crash !!
```

Eine Stringklasse mit fester Obergrenze der Länge ist keine typische Anwendung des Blockstores. Es ist ein Prinzipbeispiel, an dem wir die Funktionsweise eines Blockstores studieren können. Für das Stringproblem existiert eine noch effizientere Lösung. Der Speicherbereich fester Größe kann als Attribut direkt im Block untergebracht werden. Dafür definieren wir das folgende Klassentemplate.

```
template <class t_size, unsigned u_fixSize>
  class gct_FixBlock
    {
  public:
    typedef t_size        t_Size;
  protected:
    t_Size                o_Size;
    char                  ac_Block [u_fixSize];
  public:
    inline                gct_FixBlock ();
    inline                gct_FixBlock (const gct_FixBlock & co_init);
    inline gct_FixBlock & operator = (const gct_FixBlock & co_asgn);
    inline t_Size         GetSize () const;
    inline void           SetSize (t_Size o_newSize);
    inline void *         GetAddr () const;
    inline char *         GetCharAddr () const;
    };

template <class t_size, unsigned u_fixSize>
  inline void gct_FixBlock <t_size, u_fixSize>::
  SetSize (t_Size o_newSize)
    {
    ASSERT (o_newSize <= u_fixSize);
    o_Size = o_newSize;
    }

template <class t_size, unsigned u_fixSize>
  inline void * gct_FixBlock <t_size, u_fixSize>:: GetAddr () const
    {
    if (o_Size == 0)
      return 0;
    else
      return (void *) ac_Block;
    }
```

Dieses Template wird mit dem Größentyp und der maximalen Blockgröße parametrisiert. Eine Storeklasse muß nicht angegeben werden, denn der Speicherblock ist als Attribut im Blockobjekt enthalten und wird mit diesem erzeugt und gelöscht. Das Template `gct_FixBlock` stellt das normale Blockinterface zur Verfügung und kann als Parameter anderer Templates, die eine Blockklasse erwarten, dienen. Im folgenden Programmfragment wird mit den Parametern `unsigned char` und `8` die Blockklasse `t_FixBlock8` und damit die Stringklasse `t_FixString8` definiert. Ein Stringobjekt (zum Beispiel `o_fixString8`) umfaßt 9 Bytes.

```
typedef gct_FixBlock <unsigned char, 8> t_FixBlock8;
typedef gct_String <t_FixBlock8> t_FixString8;
....
t_FixString8 o_fixString8;
```

3.2.3 Store mit Referenzzählern

Ein Refstore realisiert eine Speicherverwaltung, die jedem Block einen Referenzzähler zuordnet. Dafür wird die Klasse ct_RefCount verwendet. Sie enthält neben dem Referenzzähler o_RefCount den Wahrheitswert b_Alloc. Die maximale Größe des Referenzzählers ist durch den Datentyp t_RefCount bestimmt. Wir definieren ihn fest auf unsigned short. Werden in einem Programm Referenzzähler unterschiedlicher Größe benötigt, muß die Klasse ct_RefCount als Template definiert werden, das den primitiven Datentyp für den Referenzzähler als Parameter erwartet. Das boolesche Attribut b_Alloc gibt Auskunft darüber, ob der Block im Refstore belegt oder frei ist. Erreicht der Referenzzähler den Wert Null und ist b_Alloc false, wird die Bedingung IsNull erfüllt.

Für eine optimale Speicherauslastung definieren wir beide Attribute der Klasse ct_RefCount als Bitfeld. Der eigentliche Referenzzähler o_RefCount umfaßt 15 Bit und kann Werte bis 32767 annehmen. Das ist für die meisten Anwendungen ausreichend. Im folgenden Programmausschnitt sehen wir die Deklaration der Klasse ct_RefCount und die Definition der Methoden DecRef und IsNull. Die Methoden enthalten nur ein oder zwei Anweisungen und sind inline definiert.

```
typedef unsigned short t_RefCount;

class ct_RefCount
  {
  t_RefCount           o_RefCount: sizeof (t_RefCount) * 8 - 1;
  bool                 b_Alloc: 1;
public:
  inline               ct_RefCount ();
  inline void          Init ();
  inline void          IncRef ();
  inline void          DecRef ();
  inline t_RefCount    GetRef () const;
  inline bool          IsAlloc () const;
  inline void          SetAlloc ();
  inline bool          IsFree () const;
  inline void          SetFree ();
  inline bool          IsNull ()const;
  };

inline void ct_RefCount:: DecRef ()
  {
  ASSERT (o_RefCount > 0);
  o_RefCount --;
  }

inline bool ct_RefCount:: IsNull ()const
  {
  return (o_RefCount == 0) && (! b_Alloc);
  }
```

Für eine hohe Flexibilität und Performance definieren wir den Refstore als ein Klassentemplate. Einziger Parameter ist der zugrunde gelegte Store. Von ihm werden die geschachtelten Typen t_Size und t_Pointer übernommen. Der Refstore nutzt die Funktionalität der übergebenen Storeklasse, erweitert sie aber nicht im Sinne einer Is-A-Relation. Deshalb erbt er nicht, sondern enthält sie als Attribut o_Store. Mit der Methode GetStore können wir darauf zugreifen. Im folgenden Programmfragment sehen wir die Deklaration des Klassentemplates. Die Methoden sind sehr einfach und können fast alle inline definiert werden, zum Beispiel MaxAlloc und StoreInfoSize.

```
template <class t_store>
  class gct_RefStore
```

```
    {
  public:
    typedef t_store:: t_Size    t_Size;
    typedef t_store:: t_Pointer t_Pointer;
  protected:
    t_store              o_Store;

    inline ct_RefCount * RefPtr (t_Pointer o_ptr) const;
  public:
    inline unsigned long MaxAlloc () const;
    inline unsigned      StoreInfoSize () const;
    t_Pointer            Alloc (t_Size o_size);
    t_Pointer            Realloc (t_Pointer o_ptr, t_Size o_size);
    inline void          Free (t_Pointer o_ptr);
    inline void *        AddrOf (t_Pointer o_ptr) const;
    inline t_Pointer     LogPtrOf (void * pv_adr) const;
    inline void          IncRef (t_Pointer o_ptr);
    inline void          DecRef (t_Pointer o_ptr);
    inline t_RefCount    GetRef (t_Pointer o_ptr) const;
    inline bool          IsAlloc (t_Pointer o_ptr) const;
    inline bool          IsFree (t_Pointer o_ptr) const;
    inline t_store *     GetStore ();
    };

template <class t_store>
  inline unsigned long gct_RefStore <t_store>:: MaxAlloc () const
    {
    return o_Store. MaxAlloc () - sizeof (ct_RefCount);
    }

template <class t_store>
  inline unsigned gct_RefStore <t_store>:: StoreInfoSize () const
    {
    return o_Store. StoreInfoSize () + sizeof (ct_RefCount);
    }
```

Die private Methode `RefPtr` wandelt die untypisierte Anfangsadresse des Blocks in einen typisierten Zeiger um. Damit können wir auf das `ct_RefCount`-Objekt zugreifen. Zum Beispiel wird in der Refstoremethode `DecRef` die Methode `ct_RefCount:: DecRef` aufgerufen. Liefert anschließend die Methode `IsNull` den Wert `true`, gibt der Refstore den Block an den darunterliegenden Store zurück. Addieren wir zum Rückgabewert von `RefPtr` mit der C++-Zeigerarithmetik den Wert Eins, erhalten wir die Adresse des nutzbaren Bereichs des Blocks. Sie wird in der Methode `AddrOf` ermittelt.

Die Adreßrechnung kann nicht wie beim Roundstore (siehe Abschnitt 3.1.1) in der Methode `Alloc` erfolgen, denn ein Zeiger des Refstores ist nicht unbedingt eine Speicheradresse. Die Refstoremethode `Alloc` muß den logischen Zeiger unverändert weitergeben. Im folgenden Abschnitt lernen wir einen Block-Refstore kennen. Seine Zeiger gehören zu einem vorzeichenlosen Zahlentyp, zum Beispiel `unsigned short`. Die Blöcke werden wie in einem Blockstore mit eins beginnend aufsteigend numeriert (siehe Abbildung im folgenden Abschnitt). Addieren wir zum logischen Zeiger Eins die Größe des `ct_RefCount`-Objekts, erhalten wir nicht die Position des nutzbaren Bereichs im Block Eins, sondern den logischen Zeiger Drei.

```
template <class t_store>
  inline ct_RefCount *
  gct_RefStore <t_store>:: RefPtr (t_Pointer o_ptr) const
    {
    ASSERT (o_ptr != 0);
    return (ct_RefCount *) o_Store. AddrOf (o_ptr);
    }
```

```
template <class t_store>
  inline void gct_RefStore <t_store>:: DecRef (t_Pointer o_ptr)
    {
    RefPtr (o_ptr)-> DecRef ();
    if (RefPtr (o_ptr)-> IsNull ())
      o_Store. Free (o_ptr);
    }

template <class t_store>
  inline void * gct_RefStore <t_store>:: AddrOf (t_Pointer o_ptr) const
    {
    if (o_ptr == 0)
      return 0;
    else
      {
      ASSERT (IsAlloc (o_ptr));
      return RefPtr (o_ptr) + 1;
      }
    }
```

Wird von einem Refstore Speicher angefordert, erhöht er die Größe um `sizeof (ct_RefCount)` Bytes und gibt die Anforderung an den darunterliegenden Store weiter. Kann dieser den Speicher bereitstellen, initialisiert der Refstore das `ct_RefCount`-Objekt und liefert den logischen Zeiger zurück. Beim Freigeben eines Blocks wird mit der Methode `ct_RefCount:: SetFree` das Attribut `b_Alloc` auf `false` gesetzt. Danach erfolgt wie bei `DecRef` die Prüfung der Bedingung `IsNull`. Ist sie erfüllt, wird der Block im darunterliegenden Store freigegeben.

```
template <class t_store>
  gct_RefStore <t_store>:: t_Pointer
  gct_RefStore <t_store>:: Alloc (t_Size o_size)
    {
    if (o_size == 0)
      return 0;
    else
      {
      t_Pointer o_ptr = o_Store. Alloc (o_size + sizeof (ct_RefCount));
      if (o_ptr == 0)
        return 0;
      else
        {
        RefPtr (o_ptr)-> Init ();
        return o_ptr;
        }
      }
    }

template <class t_store>
  inline void gct_RefStore <t_store>:: Free (t_Pointer o_ptr)
    {
    if (o_ptr != 0)
      {
      RefPtr (o_ptr)-> SetFree ();
      if (RefPtr (o_ptr)-> IsNull ())
        o_Store. Free (o_ptr);
      }
    }
```

3.2.4 Konkrete Refstores

Parametrisieren wir das Refstoretemplate mit einer Storeklasse, erhalten wir einen konkreten Refstore. Zum Beispiel ist `gct_RefStore <ct_StdStore>` ein Refstore, der auf dem Standardstore aufbaut. Mehr Flexibilität erreichen wir mit den globalen Stores, die im Makro `GLOBAL_STORE_DCLS` deklariert werden. Sie nutzen ein globales Storeobjekt und existieren in vier Versionen, die sich durch den geschachtelten Typ `t_Size` unterscheiden. Die globalen Storeklassen enthalten keine eigenen Attribute und besitzen die Größe ein Byte. Übergeben wir sie als Parameter an das Refstoretemplate, umfaßt auch der erzeugte Refstore ein Byte.

Die Klasse `gct_RefStore <ct_RndIntStore>` erweitert die Funktionalität einer globalen Roundstoreklasse um die Referenzzähler. Wir müssen beachten, daß für sie `o_ptr != AddrOf (o_ptr)` gilt. Setzen wir sie für die globale C++-Speicherverwaltung ein, müssen wir in den Operatoren `new` und `delete` logische Zeiger und Speicheradressen ineinander umrechnen. Diese Umrechnung kann bei einem Standard- oder Roundstore entfallen. Wollen wir auf den Referenzzähler eines Blocks zugreifen, müssen wir die Speicheradresse umwandeln, denn die Refstoremethoden erwarten als Parameter einen logischen Zeiger.

```
#include <stddef.h> // Für globalen Typ size_t

gct_RefStore <ct_RndIntStore> co_GlobalStore;

void * operator new (size_t u_size)
  {
  return co_GlobalStore. AddrOf (co_GlobalStore. Alloc (u_size));
  }

void operator delete (void * pv)
  {
  co_GlobalStore. Free (co_GlobalStore. LogPtrOf (pv));
  }

// operator new [] und operator delete [] analog

int main ()
  {
  char * pc = new char [20];
  co_GlobalStore. IncRef (co_GlobalStore. LogPtrOf (pc));
  ....
  }
```

Ein Round-Refstore rundet die Größe und ordnet jedem Block einen Referenzzähler zu. Diesen Komfort müssen wir mit einem erhöhten Speicherbedarf bezahlen. Am Beginn jedes Blocks speichert der Roundstore die gerundete Größe. Daran schließen sich das `ct_RefCount`-Objekt und der nutzbare Bereich des Blocks an (siehe Abbildung 3-8).

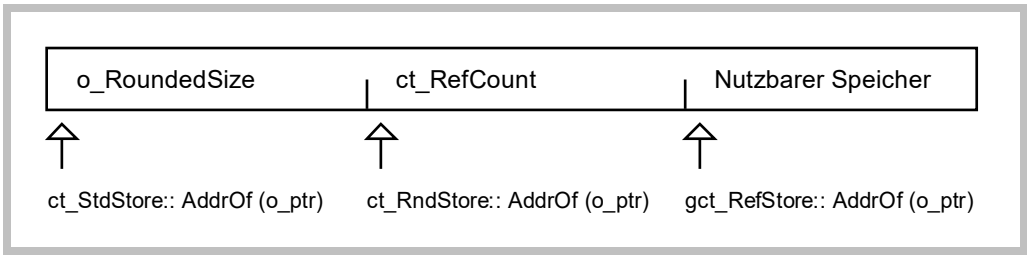

Abb. 3-8: *Ein Block im Round-Refstore*

Eine andere Art Refstores erhalten wir, wenn wir eine Blockstoreklasse als Parameter übergeben. Beim Festlegen der Blockgröße müssen wir den Referenzzähler berücksichtigen. Benötigen wir zum Beispiel einen Block-Refstore mit acht Bytes nutzbarem Speicher pro

Block, übergeben wir dem Blockstoretemplate `8 + sizeof (ct_RefCount)` als feste Blockgröße. Im folgenden Programmfragment sehen wir die dazu nötigen Definitionen. Abbildung 3-9 veranschaulicht den Aufbau eines Block-Refstores.

```
typedef gct_Block <ct_RndShortStore> t_RndShortBlock;
typedef gct_BlockStore <t_RndShortBlock, 8 + sizeof (ct_RefCount)>
  t_Fix8PlusStore;
typedef gct_RefStore <t_Fix8PlusStore> t_Fix8RefStore;
```

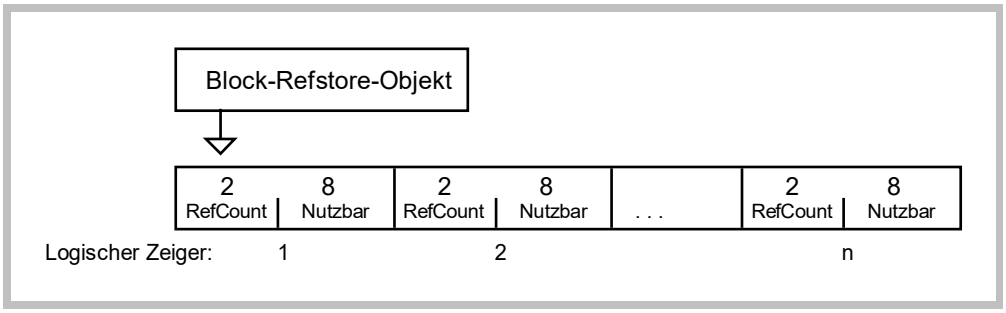

Abb. 3-9: *Aufbau eines Block-Refstores*

3.3 Neue Container braucht das Land

3.3.1 Array

Für die Implementierung der Container haben wir im Abschnitt 2.4.1 drei Vorgaben erarbeitet:

- Containerklassen erben nicht von einer abstrakten Basisklasse.
- Container werden mit Templates implementiert.
- Container enthalten die verwalteten Objekte physisch.

Die erste Bedingung ermöglicht es uns, Containerklassen ohne virtuelle Methoden zu implementieren. Damit beschleunigen sich das Iterieren und der Zugriff auf die Objekte. Container enthalten viele kleine Methoden, die `inline` definiert werden können. Bei nichtvirtuellen Inline-Methoden ist die Wahrscheinlichkeit groß, daß sie an der Verwendungsstelle vom Compiler `inline` expandiert werden. Die zweite Vorgabe sorgt für eine hohe Flexibilität. Die verwalteten Objekte müssen nicht von einer gemeinsamen Basisklasse erben. Container können somit auch für primitive Datentypen eingesetzt werden. Die dritte Bedingung ermöglicht eine optimale Speicherauslastung.

Jeder Container wird als ein Template implementiert und erwartet als ersten Parameter den Typ der Objekte, die er verwalten soll. Ein Arraycontainer kann nicht direkt auf einem Store aufbauen, denn zwischen Arrays und Stores gibt es einen wesentlichen Unterschied. In einem Store muß ein logischer Zeiger so lange seine Gültigkeit behalten, bis er mit `Realloc` oder `Free` geändert wird. Ein Array ist auf einen minimalen Speicherbedarf ausgelegt. Wird an einer bestimmten Stelle ein Objekt eingefügt oder gelöscht, werden alle nachfolgenden Objekte verschoben. Damit ändern sich deren logische Zeiger. Statt eines Stores nutzen wir einen Block als Basis für die Implementierung. Die Blockklasse wird als zweiter Parameter an das Arraytemplate übergeben.

Ein Listencontainer benötigt für jeden Eintrag ein Node. Darin befinden sich neben dem Objekt die Verweise auf Vorgänger und Nachfolger. In einem Array stehen die Objekte direkt hintereinander. Zum Verbinden der Einträge ist kein Node erforderlich. Wir benötigen jedoch

eine Hilfsklasse für das Erzeugen und Löschen der Objekte. Wir nennen das Hilfstemplate gct_ArrayNode und übergeben ihm dieselben Parameter wie dem Array. Es ist nur für den internen Gebrauch bestimmt. Alle Deklarationen sind privat, und die Arrayklasse wird als friend deklariert.

Ein Objekt wird in einem Container mit seinem Standard- oder Kopier-Konstruktor erzeugt. Für beide Fälle besitzt das Template gct_ArrayNode einen Konstruktor. Der Operator new erwartet als zweiten Parameter die Adresse des außerhalb bereitgestellten Speichers und gibt diese unverändert weiter. Er wird zum Beispiel in der Methode AddObjAfter aufgerufen (siehe unten). Der Operator delete besitzt eine leere Definition. Ein Aufruf in Operatorschreibweise führt nur zum Zerstören des Objekts mit seinem Desktruktor (siehe Methode DelObj). Die Methodenkörper sind mit Ausnahme von operator new leer. Deshalb stehen sie ausnahmsweise in der Klassendeklaration.

```
template <class t_obj, class t_block>
  class gct_ArrayNode
    {
    friend class gct_Array <t_obj, t_block>;

    t_obj                o_Obj;

    inline               gct_ArrayNode () { }
    inline               gct_ArrayNode (const t_obj & o_obj):
                           o_Obj (o_obj) { }
    static inline void * operator new (size_t, void * pv) { return pv; }
    static inline void   operator delete (void *) { }
    };
```

Neben den Implementierungsvorgaben haben wir im Abschnitt 2.4.1 auch das Interface für Container erarbeitet. Wir erweitern den Arraycontainer um die drei privaten Methoden Node, CopyFrom und DelObjects. Die Methode Node ermittelt die Adresse des Arraynodes zu einem logischen Zeiger. Sie prüft den übergebenen Zeigerwert mit zwei ASSERT-Makros und erspart uns diese Prüfungen an anderen Stellen, zum Beispiel in AddObjAfter und DelObj. CopyFrom und DelObjects enthalten die Anweisungen, die im Kopier-Konstruktor, Destruktor und Gleich-Operator gemeinsam genutzt werden. CopyFrom übernimmt die Objekte des übergebenen Arrays mit dem Kopier-Konstruktor. DelObjects ruft die Destruktoren aller enthaltenen Objekte auf. Unser Arraytemplate erbt ähnlich wie String und Blockstore privat von der übergebenen Blockklasse. Mit der Methode GetConstBlock können wir auf konstante Methoden der Basisklasse zugreifen.

Das Arraytemplate benötigt keine eigenen Attribute. Auf den dynamischen Speicherblock und dessen Größe greift es über das Blockinterface zu. Die geschachtelten Typen t_Length und t_Pointer richten sich nach dem Größentyp der Basisklasse. Wie in der Arraycollection von OHelp werden auch im Arraycontainer die Elemente mit eins beginnend aufsteigend numeriert. Im folgenden Programmausschnitt sehen wir die Deklaration des Arraytemplates und die Definition der Methoden Node und operator =.

```
template <class t_obj, class t_block>
  class gct_Array: private t_block
    {
  public:
    typedef t_block:: t_Size t_Length;
    typedef t_block:: t_Size t_Pointer;
    typedef t_obj            t_Object;
  private:
    inline gct_ArrayNode <t_obj, t_block> * Node (t_Pointer o_ptr) const;
    void                 CopyFrom (const gct_Array & co_copy);
    void                 DelObjects ();
  public:
    inline               gct_Array ();
```

```
    inline               gct_Array (const gct_Array & co_init);
    inline               ~gct_Array ();
    inline gct_Array &   operator = (const gct_Array & co_asgn);
    inline t_Length      GetLen () const;
    inline t_Pointer     First () const;
    inline t_Pointer     Next (t_Pointer o_ptr) const;
    inline t_Object *    GetObj (t_Pointer o_ptr) const;
    inline t_Pointer     AddObj (const t_Object * po_obj = 0);
    t_Pointer            AddObjCond (const t_Object * po_obj);
    t_Pointer            AddObjAfter (t_Pointer o_ptr,
                           const t_Object * po_obj = 0);
    t_Pointer            DelObj (t_Pointer o_ptr);
    inline const t_block * GetConstBlock () const;
    };

template <class t_obj, class t_block>
  inline gct_ArrayNode <t_obj, t_block> *
  gct_Array <t_obj, t_block>:: Node (t_Pointer o_ptr) const
    {
    ASSERT (o_ptr != 0);
    ASSERT (o_ptr <= GetLen ());
    return (gct_ArrayNode <t_obj, t_block> *) GetAddr () +
           (unsigned) (o_ptr - 1);
    }

template <class t_obj, class t_block>
  inline gct_Array <t_obj, t_block> &
  gct_Array <t_obj, t_block>:: operator = (const gct_Array & co_asgn)
    {
    if (& co_asgn != this)
      {
      DelObjects ();
      CopyFrom (co_asgn);
      }
    return * this;
    }
```

Mit der Methode `AddObjAfter` können wir ein neues Objekt in den Arraycontainer einfügen. Nach dem Vergrößern des Blocks wird mit der Methode `Node` die Adresse des freien Blockbereichs ermittelt. Befinden sich dahinter weitere Objekte, werden sie mit der Standardfunktion `memmove` verschoben. Dann kann das neue Objekt erzeugt werden. Dem Operator `new` unseres Hilfstemplates `gct_ArrayNode` wird die freie Adresse übergeben. Verweist der zweite Parameter der Methode `AddObjAfter` auf ein zu kopierendes Objekt, wird der Konstruktor `gct_ArrayNode (const t_obj & o_obj)` aufgerufen. Andernfalls wird das Objekt mit seinem Standard-Konstruktor initialisiert. Am Ende wird der um eins vergrößerte logische Zeiger zurückgegeben.

```
template <class t_obj, class t_block>
  gct_Array <t_obj, t_block>:: t_Pointer
  gct_Array <t_obj, t_block>:: AddObjAfter
   (t_Pointer o_ptr, const t_Object * po_obj)
    {
    SetSize (GetSize () + sizeof (gct_ArrayNode <t_obj, t_block>));
    gct_ArrayNode <t_obj, t_block> * pco_node = Node (o_ptr + 1);
    if (o_ptr < GetLen () - 1)
      memmove (pco_node + 1, pco_node, (unsigned) (GetSize () -
        (o_ptr + 1) * sizeof (gct_ArrayNode <t_obj, t_block>)));
    if (po_obj != 0)
      new (pco_node) gct_ArrayNode <t_obj, t_block> (* po_obj);
    else
      new (pco_node) gct_ArrayNode <t_obj, t_block>;
    return o_ptr + 1;
    }
```

Die Methode `DelObj` löscht ein Objekt aus dem Arraycontainer. Mit der Methode `Node` wird die Adresse des zu löschenden Nodes ermittelt. Darauf wird der `delete`-Operator des Hilfstemplates `gct_ArrayNode` angewendet. Befinden sich dahinter weitere Objekte, werden sie um eine Position verschoben. Anschließend wird der Block verkleinert und der logische Zeiger des Nachfolgers zurückgegeben.

```
template <class t_obj, class t_block>
  gct_Array <t_obj, t_block>:: t_Pointer
  gct_Array <t_obj, t_block>:: DelObj (t_Pointer o_ptr)
    {
    gct_ArrayNode <t_obj, t_block> * pco_node = Node (o_ptr);
    delete pco_node;
    if (o_ptr < GetLen ())
      memmove (pco_node, pco_node + 1, (unsigned) (GetSize () -
        o_ptr * sizeof (gct_ArrayNode <t_obj, t_block>)));
    SetSize (GetSize () - sizeof (gct_ArrayNode <t_obj, t_block>));
    return Next (o_ptr - 1);
    }
```

Das Speicherlayout eines Arraycontainers hängt vor allem vom Templateparameter `t_block` ab. Im einfachsten Fall nutzen wir eine Blockklasse, die ihren Speicher von einem globalen Store anfordert. In Abbildung 3-10 sehen wir das Layout einer Instanz des Typs `gct_Array <ct_AnyClass, gct_Block <ct_StdShortStore> >`.

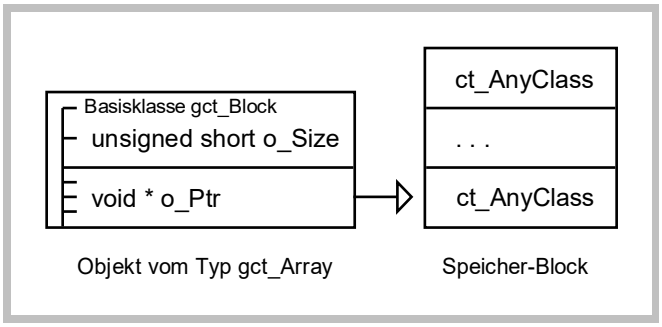

Abb. 3-10: *Speicherlayout eines Arraycontainers*

3.3.2 DList

Unser Listencontainer stellt die Funktionalität einer doppelt verketteten Liste zur Verfügung. Für jeden Eintrag benötigt er ein Node. Dieses übt eine doppelte Funktion aus. Zum einen enthält es die Verweise auf Vorgänger und Nachfolger. Zum anderen besitzt es eigene Operatoren `new` und `delete`, mit deren Hilfe das Objekt erzeugt und gelöscht wird. Als Parameter werden dem Template `gct_DListNode` der Objekt- und der Zeigertyp übergeben. In einer Blockliste (siehe folgenden Abschnitt) wird das Node zum Parametrisieren des Storetemplates benötigt. Die Storeklasse ist dem Node nicht bekannt und somit auch nicht der genaue Listentyp, zu dem es gehört. Die Liste kann nicht als `friend` deklariert werden. Folglich müssen alle Deklarationen `public` sein.

```
template <class t_obj, class t_ptr>
  class gct_DListNode
    {
  public:
    t_ptr               o_Prev;
    t_ptr               o_Next;
    t_obj               o_Obj;
```

```
    inline              gct_DListNode () { }
    inline              gct_DListNode (const t_obj & o_obj):
                          o_Obj (o_obj) { }
    static inline void * operator new (size_t, void * pv) { return pv; }
    static inline void   operator delete (void *) { }
    };
```

Der Listencontainer `gct_DList` baut direkt auf einem Store auf. Die Storeklasse wird als zweiter Templateparameter übergeben. Von ihr werden Längen- und Zeigertyp übernommen, und sie ist als Attribut im Listenobjekt enthalten. Auf das Attribut `o_Store` können wir mit der Methode `GetStore` zugreifen. Neben dem allgemeinen Containerinterface enthält das Listentemplate einige private Methoden. `Node` liefert zu einem logischen Zeiger die Adresse des Nodes. Die Methode `NewNode` erzeugt ein neues Node und aktualisiert die Verweise der benachbarten Nodes. `CopyFrom` kopiert alle Nodes einer anderen Liste und `ClearList` löscht alle Nodes. Im folgenden Programmausschnitt sehen wir die Deklaration des Listentemplates und die Definition der Methoden `Node` und `operator =`.

```
template <class t_obj, class t_store>
  class gct_DList
    {
  public:
    typedef t_store:: t_Size    t_Length;
    typedef t_store:: t_Pointer t_Pointer;
    typedef t_obj               t_Object;
  protected:
    t_Length            o_Length;
    t_Pointer           o_First;
    t_store             o_Store;

    inline gct_DListNode <t_Object, t_Pointer> *
                        Node (t_Pointer o_ptr) const;
    t_Pointer           NewNode (t_Pointer, t_Pointer, const t_obj *);
    void                CopyFrom (const gct_DList & co_copy);
    void                ClearList ();
  public:
    inline              gct_DList ();
    inline              gct_DList (const gct_DList & co_init);
    inline              ~gct_DList ();
    inline gct_DList &  operator = (const gct_DList & co_asgn);
    inline t_Length     GetLen () const;
    inline t_Pointer    First () const;
    inline t_Pointer    Next (t_Pointer o_ptr) const;
    inline t_Object *   GetObj (t_Pointer o_ptr) const;
    inline t_Pointer    AddObj (const t_Object * po_obj = 0);
    t_Pointer           AddObjCond (const t_Object * po_obj);
    t_Pointer           AddObjAfter (t_Pointer o_ptr,
                          const t_Object * po_obj = 0);
    t_Pointer           DelObj (t_Pointer o_ptr);
    inline t_store *    GetStore ();
    };

template <class t_obj, class t_store>
  inline gct_DListNode <t_obj, gct_DList <t_obj, t_store>:: t_Pointer> *
  gct_DList <t_obj, t_store>:: Node (t_Pointer o_ptr) const
    {
    ASSERT (o_ptr != 0);
    return (gct_DListNode <t_obj, t_Pointer> *) o_Store. AddrOf (o_ptr);
    }

template <class t_obj, class t_store>
  inline gct_DList <t_obj, t_store> &
  gct_DList <t_obj, t_store>:: operator = (const gct_DList & co_asgn)
    {
```

```
    if (& co_asgn != this)
      {
      ClearList ();
      CopyFrom (co_asgn);
      }
    return * this;
    }
```

Der Methode `NewNode` werden drei Parameter übergeben: Die logischen Zeiger des Vorgängers und Nachfolgers und ein C++-Zeiger auf das zu kopierende Objekt. Am Anfang wird vom Store der Speicher für ein neues Node angefordert. Der logische Zeiger wird in eine Adresse umgewandelt und dem Operator `new` der Nodeklasse übergeben. In Abhängigkeit vom Wert des Parameters `po_obj` wird das neue Objekt im Node mit seinem Kopier- oder Standard-Konstruktor erzeugt. Danach wird das neue Node mit seinem Vorgänger und Nachfolger verbunden. Ist die Liste leer, sind Vorgänger- und Nachfolgerzeiger gleich Null. In diesem Fall muß es mit sich selbst verbunden werden. Der logische Zeiger des neuen Nodes ist zugleich der Rückgabewert der Methode `NewNode`.

```
template <class t_obj, class t_store>
  gct_DList <t_obj, t_store>:: t_Pointer
  gct_DList <t_obj, t_store>:: NewNode
   (t_Pointer o_prev, t_Pointer o_next, const t_obj * po_obj)
    {
    gct_DListNode <t_Object, t_Pointer> * pco_node;
    t_Pointer o_new =
      o_Store. Alloc (sizeof (gct_DListNode <t_Object, t_Pointer>));
    ASSERT (o_new != 0);
    void * pv = o_Store. AddrOf (o_new);
    if (po_obj != 0)
      pco_node = new (pv) gct_DListNode <t_Object, t_Pointer> (* po_obj);
    else
      pco_node = new (pv) gct_DListNode <t_Object, t_Pointer>;
    if (o_prev != 0)
      {
      pco_node-> o_Prev = o_prev;
      pco_node-> o_Next = o_next;
      Node (o_prev)-> o_Next = o_new;
      Node (o_next)-> o_Prev = o_new;
      }
    else
      pco_node-> o_Prev = pco_node-> o_Next = o_new;
    return o_new;
    }
```

Die Methode `AddObjAfter` eines Containers muß für beide Parameter den Wert Null akzeptieren. Ist der logische Zeiger des Objekts, hinter dem eingefügt werden soll, gleich Null, wird das neue Objekt an den Anfang des Containers gestellt. Der Algorithmus in `gct_Array:: AddObjAfter` arbeitet auch für den Zeigerwert Null korrekt. Im Listencontainer muß dieser Fall gesondert behandelt werden. Eine weitere Fallunterscheidung ist für eine leere Liste nötig.

```
template <class t_obj, class t_store>
  gct_DList <t_obj, t_store>:: t_Pointer
  gct_DList <t_obj, t_store>:: AddObjAfter
   (t_Pointer o_ptr, const t_Object * po_obj)
    {
    o_Length ++;
    if (o_ptr != 0)
      return NewNode (o_ptr, Node (o_ptr)-> o_Next, po_obj);
    else
      if (o_First != 0)
        return
```

```
        o_First = NewNode (Node (o_First)-> o_Prev, o_First, po_obj);
      else
        return o_First = NewNode (0, 0, po_obj);
    }
```

Die Methode `DelObj` aktualisiert die Attribute `o_Length` und `o_First`. Danach wird der Zeiger auf das zu löschende Node ermittelt. Es wird aus der Verweiskette der Liste herausgelöst, indem Vorgänger und Nachfolger miteinander verbunden werden. Anschließend wird mit dem Operator `delete` der Destruktor des Nodes aufgerufen und mit der Storemethode `Free` der belegte Speicher freigegeben.

```
template <class t_obj, class t_store>
  gct_DList <t_obj, t_store>:: t_Pointer
  gct_DList <t_obj, t_store>:: DelObj (t_Pointer o_ptr)
    {
    ASSERT (o_ptr != 0);
    ASSERT (o_Length != 0);
    o_Length --;
    t_Pointer o_next = Next (o_ptr);
    if (o_ptr == o_First)
      o_First = o_next;
    gct_DListNode <t_Object, t_Pointer> * po_node = Node (o_ptr);
    Node (po_node-> o_Prev)-> o_Next = po_node-> o_Next;
    Node (po_node-> o_Next)-> o_Prev = po_node-> o_Prev;
    delete po_node;
    o_Store. Free (o_ptr);
    return o_next;
    }
```

Das Speicherlayout eines Listencontainers hängt vor allem von der übergebenen Storeklasse ab. Im einfachsten Fall nutzen wir für den Templateparameter `t_store` eine globale Storeklasse. In Abbildung 3-11 sehen wir das Layout einer Instanz des Typs `gct_DList <ct_AnyClass, ct_StdShortStore>`. Das Listenobjekt umfaßt sieben Bytes. Die Storeklasse ist mit einem Dummybyte an der Größe des Objekts beteiligt.

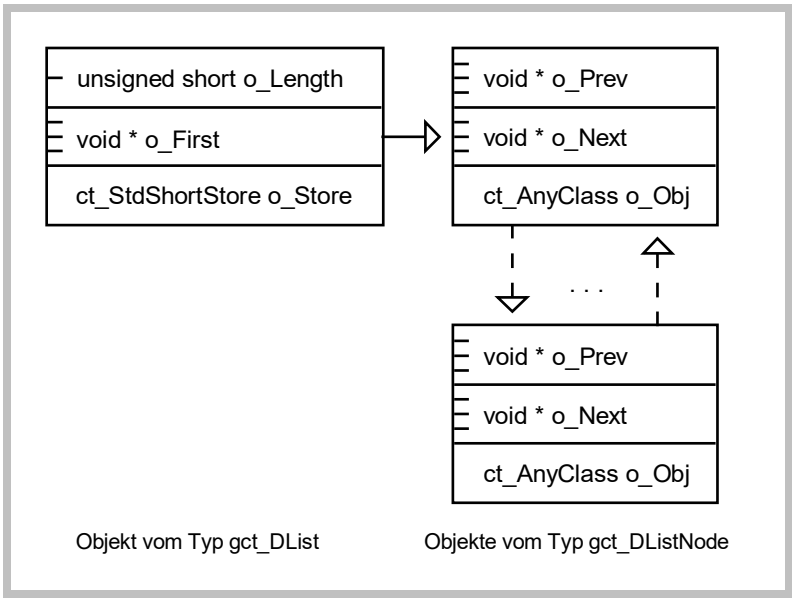

Abb. 3-11: *Speicherlayout eines Listencontainers*

Ein Listencontainer stellt sicher, daß die *logischen Zeiger* der enthaltenen Objekte ihre Gültigkeit behalten. Im allgemeinen bleibt jedoch die *Adresse* eines Objekts nicht konstant. Der Container kann es an eine andere Stelle im Speicher verschieben. Die Gültigkeitsdauer der Adressen in einem Listencontainer hängt von der übergebenen Storeklasse ab. Nutzen

wir eine globale Storeklasse (wie oben), bleiben auch die Adressen der Objekte konstant. Die Speicherauslastung ist dabei nicht so gut, denn jedes Node beansprucht einen eigenen Block der dynamischen Speicherverwaltung.

3.3.3 Block- und Reflisten

Eine bessere Speicherauslastung erzielen wir mit einem Listencontainer, der auf einem Blockstore aufbaut. Eine **Blockliste** speichert alle Nodes in einem einzigen Block. Dadurch wird die dynamische Speicherverwaltung entlastet. Den Speicherbedarf können wir mit geeigneten Zeigertypen reduzieren. Jedes Node enthält zwei Zeiger. Verwenden wir statt `unsigned long` den Zeigertyp `unsigned short`, beanspruchen die Zeiger nur noch halb so viel Speicher (siehe Abbildung 3-12). Bei der Arbeit mit Blocklisten müssen wir beachten, daß sich die Adressen der Objekte ändern können, sobald ein Objekt eingefügt oder gelöscht wird.

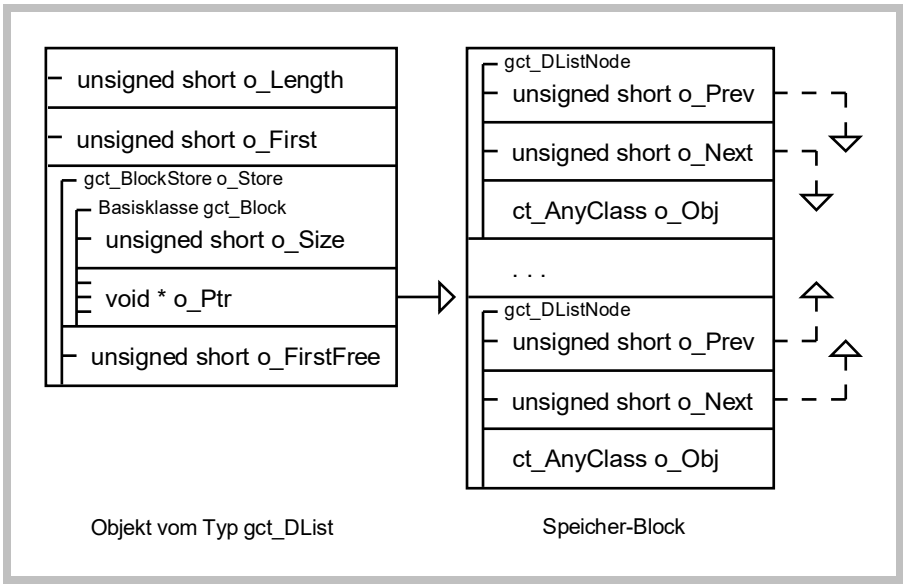

Abb. 3-12: *Speicherlayout einer Blockliste*

Zum Definieren einer Blockliste sind mehrere Schritte erforderlich. Als Grundlage benötigen wir eine Blockklasse. Mit der Objektklasse und dem Größentyp der Blockklasse können wir das Listennode parametrisieren und seine Größe berechnen. Diese wird als zweiter Parameter an das Blockstoretemplate übergeben. Mit der Objektklasse und der Storeklasse können wir nun die Containerklasse erzeugen. Das Speicherlayout der Blockliste, die im folgenden Programmfragment definiert wird, stimmt mit Abbildung 3-12 überein.

```
typedef gct_Block <ct_StdShortStore> t_StdShortBlock;
const int i_FixSize =
  sizeof (gct_DListNode <ct_AnyClass, t_StdShortBlock:: t_Size>);
typedef gct_BlockStore <t_StdShortBlock, i_FixSize> t_StdShortBlockStore;
typedef gct_DList <ct_AnyClass, t_StdShortBlockStore> t_StdShortBlockList;
```

Parametrisieren wir das Listentemplate mit einem Refstore, wird jedem Node ein Referenzzähler zugeordnet. Eine **Refliste** kann überall dort eingesetzt werden, wo Einträge der Liste von außerhalb referenziert werden. Diese Referenzen werden bei Veränderungen der Liste keine Dangling Pointers. Vor einem Zugriff kann die Liste mit der Methode `IsAlloc` gefragt werden, ob sie den Eintrag noch besitzt. Wurde er inzwischen gelöscht, sollte die externe Referenz den Referenzzähler verkleinern, damit der Speicher des Listeneintrags freigegeben werden kann.

Der Zugriff auf die Referenzzähler erfolgt über das Storeobjekt der Liste. Dazu müßte jedesmal die Methode `GetStore` aufgerufen werden. Zur Vereinfachung definieren wir das Template `gct_RefDList`. Es erweitert das Listeninterface um die Zugriffsmethoden auf den Referenzzähler. Der Templateparameter `t_store` muß eine Refstoreklasse sein.

```
template <class t_obj, class t_store>
  class gct_RefDList: public gct_DList <t_obj, t_store>
    {
  public:
    inline void          IncRef (t_Pointer o_ptr);
    inline void          DecRef (t_Pointer o_ptr);
    inline t_RefCount    GetRef (t_Pointer o_ptr) const;
    inline bool          IsAlloc (t_Pointer o_ptr) const;
    inline bool          IsFree (t_Pointer o_ptr) const;
    };

template <class t_obj, class t_store>
  inline bool
  gct_RefDList <t_obj, t_store>:: IsAlloc (t_Pointer o_ptr) const
  {
  return o_Store. IsAlloc (o_ptr);
  }
```

Im folgenden Programmfragment wird eine Refliste definiert und in einem Prinzipbeispiel angewendet. Sie fordert den Speicher ihrer Nodes vom globalen Standardstore an. Ihr Längentyp ist `unsigned short` und ihr Zeigertyp `void *`. Jedes Node belegt einen eigenen Speicherblock. Folglich bleiben die Adressen der enthaltenen Objekte konstant. Das Speicherlayout dieser Liste ist in Abbildung 3-13 zu sehen.

```
typedef gct_RefDList <ct_AnyClass, gct_RefStore <ct_StdShortStore> >
  t_StdShortRefList;
t_StdShortRefList co_list;
t_StdShortRefList:: t_Pointer o_ptr = co_list. AddObj ();
co_list. IncRef (o_ptr);
if (co_list. IsAlloc (o_ptr))
  co_list. GetObj (o_ptr)-> ....;  // Wird ausgeführt
co_list. DelObj (o_ptr);
if (co_list. IsAlloc (o_ptr))
  co_list. GetObj (o_ptr)-> ....;  // Wird nicht ausgeführt
else
  {
  co_list. DecRef (o_ptr);  // Wird ausgeführt
  o_ptr = 0;
  }
```

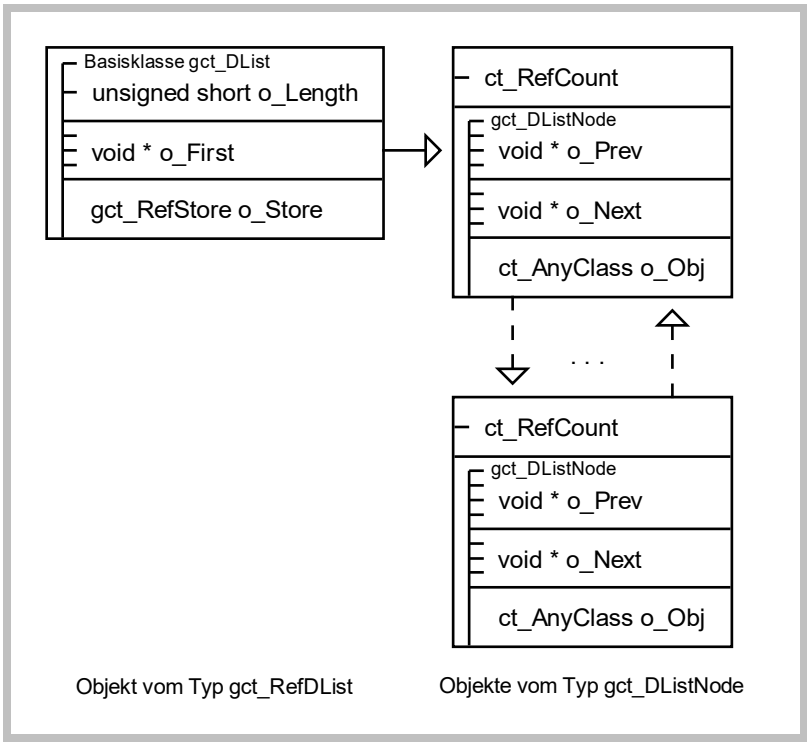

Abb. 3-13: *Speicherlayout einer Refliste*

Eine Block-Refliste verbindet die Sicherheit einer Refliste mit der guten Speicherauslastung einer Blockliste. Der Listentyp wird ähnlich wie eine Blockliste definiert. Bei der festen Blockgröße muß das `ct_RefCount`-Objekt berücksichtigt werden, das jedem Block zugeordnet wird. Abbildung 3-14 zeigt das Speicherlayout der Block-Refliste, die im folgenden Programmfragment definiert wird.

```
typedef gct_Block <ct_StdShortStore> t_StdShortBlock;
const int i_FixSize = sizeof (ct_RefCount) +
  sizeof (gct_DListNode <ct_AnyClass, t_StdShortBlock:: t_Size>);
typedef gct_BlockStore <t_StdShortBlock, i_FixSize> t_StdShortBlockStore;
typedef gct_RefDList <ct_AnyClass, gct_RefStore <t_StdShortBlockStore> >
  t_StdShortBlockRefList;
```

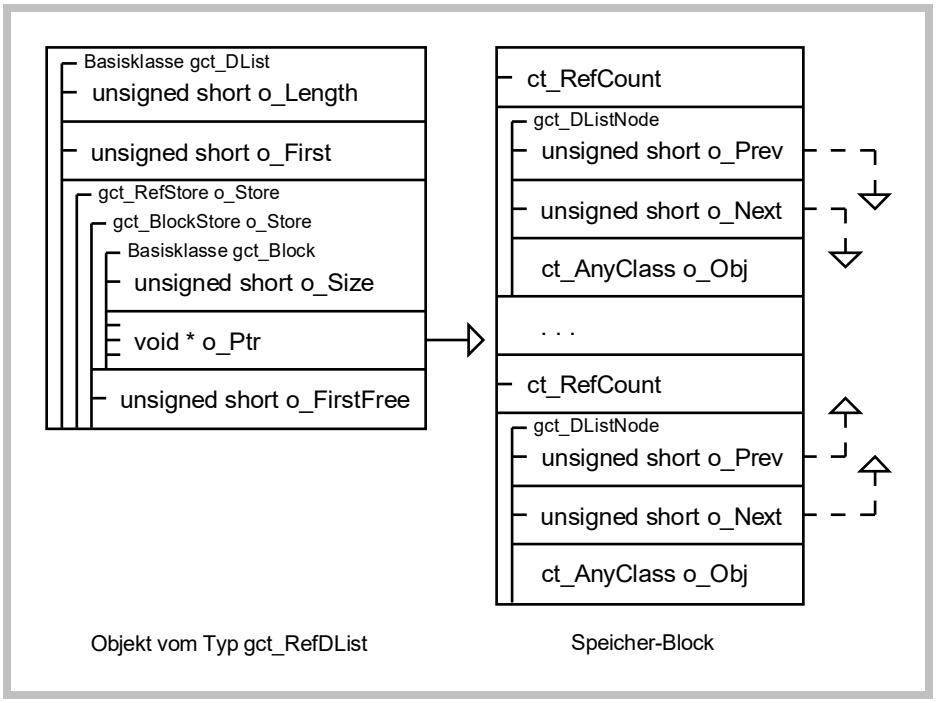

Abb. 3-14: *Speicherlayout einer Block-Refliste*

Diese Block-Refliste ist die komplizierteste Klasse, die wir bisher kennengelernt haben. Würden wir keine zusätzlichen Typdefinitionen verwenden, müßten wir schreiben:
`gct_RefDList <ct_AnyClass, gct_RefStore <gct_BlockStore <gct_Block <ct_StdShortStore>, sizeof (ct_RefCount) + sizeof (gct_DListNode <ct_AnyClass, gct_Block <ct_StdShortStore>:: t_Size>)> > >`.
Abbildung 3-15 bringt Übersicht in diesen Dschungel geschachtelter Templates. Das Designdiagramm enthält alle beteiligten Klassen mit ihren wichtigsten Verbindungen. Um die Namen der Templateklassen nicht zu lang werden zu lassen, werden gegenüber dem Programmfragment die folgenden Abkürzungen verwendet.

- i_Size statt i_FixSize
- ct_Cls statt ct_AnyClass
- t_ShBlk statt t_StdShortBlock
- t_ShBlkSize statt t_StdShortBlock:: t_Size
- t_ShBlkSt statt t_StdShortBlockStore
- t_ShBlkRefSt statt gct_RefStore <t_StdShortBlockStore>

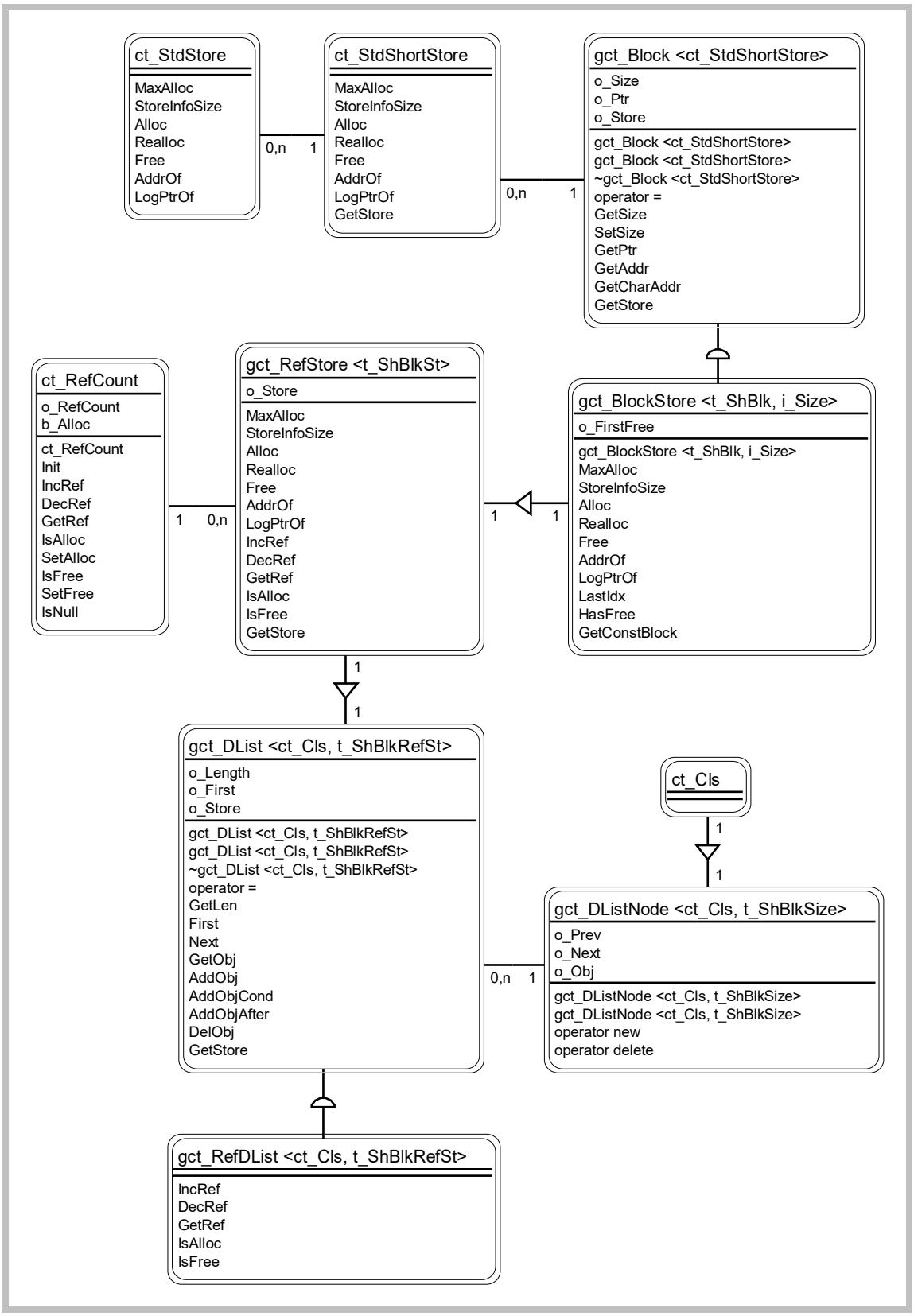

Abb. 3-15: *Verwendete Klassen der Block-Refliste*

Im Designdiagramm erscheinen nichtstatische Attribute als Teil-Ganzes-Beziehung (Dreieck). Andere Abhängigkeiten zwischen Objekten werden als Objekt-Verbindung dargestellt (durchgehende Linie). Das Fundament bildet die Klasse `ct_StdStore`. Von ihr werden in den `GLOBAL_STORE`-Makros eine globale Instanz und vier globale Klassen generiert, darunter

ct_StdShortStore. Alle Instanzen dieser Storeklasse greifen auf denselben globalen Standardstore zu. ct_StdShortStore ist als statisches Attribut in t_StdShortBlock enthalten. Der Blockstore erbt von der Blockklasse und ist Teil des Refstores. Der Refstore verwaltet null bis n ct_RefCount-Objekte und ist als Attribut o_Store in der Liste enthalten. Diese verwaltet null bis n Nodes, in denen sich je ein ct_AnyClass-Objekt befindet. Als letzte Klasse in dieser Reihe erbt die Refliste von der normalen Liste.

Am Ende des Abschnitts 2.4.1 haben wir verschiedene Containerarten in bezug auf die Gültigkeitsdauer ihrer logischen Zeiger und der Adressen ihrer Objekte miteinander verglichen. Inzwischen hat sich das Spektrum der Container erweitert. In Tabelle 3-4 sind die wichtigsten Arten zusammengefaßt. Reflisten und Block-Reflisten betrachten wir in diesem Zusammenhang nicht. Sie verhalten sich bezüglich der Gültigkeitsdauer wie die Basislisten, von denen sie erben.

Containerart	Beispiel
Objekt-Array	gct_Array <float, gct_Block <ct_StdShortStore> >
Zeiger-Array	gct_Array <float *, gct_Block <ct_StdShortStore> >
Objekt-Liste	gct_DList <float, ct_StdShortStore>
Zeiger-Liste	gct_DList <float *, ct_StdShortStore>
Objekt-Blockliste	gct_DList <float, t_StdShortBlockStore>
Zeiger-Blockliste	gct_DList <float *, t_StdShortBlockStore>

Tab. 3-4: *Containerarten*

In Tabelle 3-5 sind diese Containerarten nach der Gültigkeitsdauer der logischen Zeiger und Adressen geordnet. Die erste Spalte enthält Container, deren logische Zeiger nach einer Änderung (Einfügen oder Löschen von Elementen) ungültig werden können. Dazu zählen nur Arraycontainer. In der zweiten Spalte finden wir ausschließlich Listencontainer. Die obere Zeile der Tabelle enthält Container, die ihre Objekte im Speicher verschieben. Dabei ändern sich zwar die Adressen der Objekte, der belegte Speicher wird jedoch optimal ausgelastet. In der unteren Zeile befinden sich Container, deren Objekte stets an derselben Stelle bleiben. Diese Container benötigen mehr Speicher.

	Logischer Zeiger ungültig	Logischer Zeiger gültig
Adresse ungültig	Objekt-Array	Objekt-Blockliste
Adresse gültig	Zeiger-Array	Objekt-L., Zeiger-L., Zeiger-Blockl.

Tab. 3-5: *Logische Zeiger und Adressen in Containern*

3.3.4 Test der Container

Container und Collections müssen wir sorgfältig testen. Sie werden an zahlreichen Stellen eingesetzt. Tritt in einem Programm ein Fehler auf, vermuten wir ihn zunächst in den eigenen Klassen. Von den fundamentalen Klassen erwarten wir Korrektheit und Robustheit. Bei Containern ist der Testaufwand höher als bei Collections. Neben den Methoden zum Einfügen, Löschen und Iterieren müssen wir auch prüfen, ob die Konstruktoren und Destruktoren der enthaltenen Objekte korrekt aufgerufen werden. Zu diesem Zweck verwenden wir die folgende Testklasse. Sie enthält alle Methoden, die in einem Container direkt oder indirekt aufgerufen werden. Jede Methode protokolliert eine entsprechende Meldung auf die Standardausgabe.

```
class ct_Int: public ct_Object
  {
  int                     i_Value;
public:
                          ct_Int ();
                          ct_Int (int i);
                          ct_Int (const ct_Int & co_int);
  virtual                 ~ct_Int ();
  ct_Int &                operator = (int i);
  ct_Int &                operator = (const ct_Int & co_int);
  virtual const char *    GetTypeName () const;
  bool                    operator == (const ct_Int & co_int);
  int                     GetValue () const { return i_Value; }
  };

ct_Int:: ct_Int (const ct_Int & co_int)
  {
  i_Value = co_int. i_Value;
  printf ("ct_Int (const ct_Int & co_int = %d)\n", i_Value);
  }

ct_Int & ct_Int:: operator = (int i)
  {
  i_Value = i;
  printf ("operator = (int i = %d)\n", i_Value);
  return * this;
  }
```

Container erben nicht von einer gemeinsamen Basisklasse. Zum Protokollieren ihres Inhalts können wir keine normale Funktion verwenden. Stattdessen definieren wir das folgende Funktionstemplate. Es erwartet als Funktionsparameter einen Zeiger auf einen unbekannten Container, der `ct_Int`-Objekte enthält. Von jedem Eintrag wird der logische Zeiger und der Wert ausgegeben.

```
template <class t_container>
  void PrintContainer (t_container * po_cont)
    {
    printf ("Container:");
    for (t_container:: t_Pointer o_ptr = po_cont-> First ();
         o_ptr != 0;
         o_ptr = po_cont-> Next (o_ptr))
      {
      printf (" Entry[%ld]=%d",
        (long) o_ptr, po_cont-> GetObj (o_ptr)-> GetValue ());
      }
    printf ("\n");
    }
```

Das folgende Programmfragment enthält einige wichtige Tests für Container. Es erhebt keinen Anspruch auf Vollständigkeit, ist aber eine gute Grundlage für ein umfangreicheres Testprogramm. Am Anfang wird die Methode `AddObjCond` geprüft. Beim zweiten Aufruf mit demselben Objekt darf sie es nicht noch einmal einfügen. Danach wird die Methode `AddObjAfter` mit logischen Zeigern ungleich und gleich Null aufgerufen. Als Ergebnis müssen sich drei Objekte mit den Werten 0, 1 und 2 (in dieser Reihenfolge) im Container befinden. Der nächste Abschnitt prüft den Kopier-Konstruktor, Gleich-Operator und Destruktor eines kompletten Containers. Ein zweiter Container wird erzeugt. Nach dem Zuweisen eines neuen Inhalts wird er wieder zerstört. Am Ende werden aus dem ursprünglichen Container zwei Objekte gelöscht, und auch dieser wird zerstört.

```
typedef .... <ct_Int> t_container;
ct_Int co_int (1);
```

```
t_container * po_cont = new t_container;
PrintContainer (po_cont);
t_container:: t_Pointer o_1 = po_cont-> AddObjCond (& co_int);
po_cont-> AddObjCond (& co_int);
PrintContainer (po_cont);

co_int = 2;
o_1 = po_cont-> AddObjAfter (o_1, & co_int);
PrintContainer (po_cont);
co_int = 0;
po_cont-> AddObjAfter (0, & co_int);
PrintContainer (po_cont);

printf ("Beginn zweiter Container\n");
t_container * po_cont2 = new t_container (* po_cont);
PrintContainer (po_cont2);
* po_cont = * po_cont2;
PrintContainer (po_cont);
delete po_cont2;
printf ("Ende zweiter Container\n");

po_cont-> DelObj (o_1);
PrintContainer (po_cont);
po_cont-> DelObj (po_cont-> First ());
PrintContainer (po_cont);
delete po_cont;
```

Das Protokoll des Testprogramms muß für alle Container gleich sein. Nur die logischen Zeiger dürfen sich unterscheiden. Bei der Analyse des folgenden Textes müssen wir beachten, daß die Variable `co_int` am Anfang erzeugt und am Ende zerstört wird. Die Zusätze in C++-Kommentarform stammen vom Autor.

```
ct_Int (int i = 1)                    // co_int (1)
Container:
ct_Int (const ct_Int & co_int = 1)    // AddObjCond
Container: Entry[1]=1
operator = (int i = 2)                // co_int = 2
ct_Int (const ct_Int & co_int = 2)    // AddObjAfter
Container: Entry[1]=1 Entry[2]=2
operator = (int i = 0)                // co_int = 0
ct_Int (const ct_Int & co_int = 0)    // AddObjAfter
Container: Entry[3]=0 Entry[1]=1 Entry[2]=2
Beginn zweiter Container
ct_Int (const ct_Int & co_int = 0)    \
ct_Int (const ct_Int & co_int = 1)     >  // Kopier-Konstruktor
ct_Int (const ct_Int & co_int = 2)    /
Container: Entry[1]=0 Entry[2]=1 Entry[3]=2
~ct_Int (0)                           \
~ct_Int (1)                            \
~ct_Int (2)                             \  // Gleich-Operator
ct_Int (const ct_Int & co_int = 0)      /  // des Containers
ct_Int (const ct_Int & co_int = 1)     /
ct_Int (const ct_Int & co_int = 2)    /
Container: Entry[1]=0 Entry[2]=1 Entry[3]=2
~ct_Int (0)                           \
~ct_Int (1)                            >  // Destruktor
~ct_Int (2)                           /
Ende zweiter Container
~ct_Int (1)                           // DelObj
Container: Entry[1]=0 Entry[3]=2
~ct_Int (0)                           // DelObj
Container: Entry[3]=2
~ct_Int (2)                           // Destruktor
```

```
~ct_Int (0)                          // co_int
```

Wer das Buch aufmerksam gelesen hat, sieht auf den ersten Blick, daß eine Blockliste getestet wurde. Die logischen Zeiger besitzen die Werte 1, 2 und 3. Dafür kommen nur Arrays und Blocklisten infrage. Beim Durchlaufen des Containers nach `AddObjAfter` erscheinen sie in der Reihenfolge 3, 1 und 2. Damit ist ein Array ausgeschlossen. Im folgenden Text sehen wir dieselbe Zeile aus dem Protokoll eines Arrays und einer normalen Liste.

```
// Array
Container: Entry[1]=0 Entry[2]=1 Entry[3]=2
// Normale Liste
Container: Entry[898367494]=0 Entry[898105350]=1 Entry[898236422]=2
```

3.4 Griff ins Regal

Die Implementierung der Container mit Hilfe von Templates führt nicht nur zu einer guten Performance, sondern auch zu einer hohen Flexibilität. Containertemplates lassen sich an ein breites Spektrum von Rahmenbedingungen anpassen. Wir lernten bisher vier konkrete Listencontainer kennen, die ein unterschiedliches Speicher- und Rechenzeitverhalten aufweisen. Sie wurden aus einem einzigen Listentemplate generiert und besitzen dasselbe Interface. Damit ist ihre Handhabung einfach. Ein Container läßt sich leicht durch einen anderen ersetzen.

Zum Definieren einer Containerklasse sind jedoch mehrere Schritte erforderlich. Diese Teilschritte, zum Beispiel das Berechnen der Größe eines Listennodes, sind für den Anwender beschwerlich und fehleranfällig. Um die Handhabung zu vereinfachen, werden im folgenden einige *Instanzen vordefiniert*. Diese ersparen umständliche Typdefinitionen und ermöglichen es, mit einem einzigen Griff ins Regal den passenden Container zu finden.

3.4.1 Vordefinierte Stores und Blöcke

Container werden auf der Grundlage von Stores und Blöcken gebildet. Bevor wir mit den Containern beginnen, generieren wir Instanzen der Speicherverwaltungsklassen. Dabei orientieren wir uns an den globalen Stores (siehe Abschnitt 3.1.2). Diese werden in Präprozessormakros deklariert und besitzen generierte Namen. Das Makro `GLOBAL_STORE_DCL` deklariert eine Storeklasse für einen bestimmten Größentyp. Im Makro `GLOBAL_STORE_DCLS` werden Storeklassen für vier Größentypen gebildet.

`GLOBAL_STORE_DCL` besitzt vier Parameter. `t_store` bezeichnet die Storeklasse, von der eine globale Instanz gebildet werden soll, zum Beispiel `ct_StdStore` oder `ct_RndStore`. Der Parameter `Obj` enthält eine Identität für das globale Objekt. Vordefiniert sind die Objektidentitäten `Std` und `Rnd`. Daraus werden die globalen Storeobjekte `co_StdStore` und `co_RndStore` generiert. Der dritte Parameter `Size` enthält eine Kurzbezeichnung des geschachtelten Größentyps `t_Size`. Sie wird für die Namensbildung benötigt. Es steht `Int` für `unsigned int`, `Char` für `unsigned char` usw. Der letzte Parameter `t_size` enthält den zugehörigen C++-Typ. Zum Beispiel wird in `GLOBAL_STORE_DCL (ct_StdStore, Std, Long, unsigned long)` die globale Storeklasse `ct_StdLongStore` mit dem geschachtelten Größentyp `unsigned long` deklariert.

Für vordefinierte Instanzen des Blocktemplates benötigen wir nur die Parameter `Obj` und `Size`. Zur Vereinfachung erwartet das Makro `BLOCK_DCL` beide Angaben als einen zusammenhängenden Bezeichner mit dem Präfix `ct_`. Zum Beispiel wird in `BLOCK_DCL`

(ct_StdLong) die Blockklasse ct_StdLongBlock deklariert. Sie enthält den Größentyp unsigned long und fordert ihren Speicher vom globalen Storeobjekt co_StdStore an.

```
#define BLOCK_DCL(StoreSpec)                    \
  class StoreSpec ## Block:                     \
    public gct_Block <StoreSpec ## Store> { };

// Beispiel: BLOCK_DCL (ct_StdLong) expandiert zu
class ct_StdLongBlock:
  public gct_Block <ct_StdLongStore> { };
```

Das Blockstoretemplate besitzt zwei Parameter, die Blockklasse t_block und die feste Größe der Blöcke u_fixSize. Der zweite Parameter bleibt auch bei vordefinierten Instanzen variabel. Diese sind also keine Klassen, sondern wieder Templates. Sie besitzen aber nur noch den Parameter u_fixSize. Für jeden Größentyp werden zwei Blockstoretemplates gebildet. Das zweite ist ein Block-Refstore. Die Größe des ct_RefCount-Objekts wird im Makro berücksichtigt.

Das Makro BLOCK_STORE_DCL erwartet einen Parameter in derselben Form wie BLOCK_DCL. Das darin erzeugte Blockstoretemplate verwendet eine generierte Blockklasse. Zum Beispiel wird in BLOCK_STORE_DCL (ct_RndShort) das Template gct_RndShortBlockStore generiert. Es verwendet die Klasse ct_RndShortBlock. Das Block-Refstoretemplate baut auf dem im selben Makro erzeugten Blockstoretemplate auf.

```
#define BLOCK_STORE_DCL(StoreSpec)                              \
  template <unsigned u_fixSize>                                 \
    class g ## StoreSpec ## BlockStore:                         \
      public gct_BlockStore <StoreSpec ## Block, u_fixSize> { }; \
  template <unsigned u_fixSize>                                 \
    class g ## StoreSpec ## BlockRefStore:                      \
      public gct_RefStore <g ## StoreSpec ## BlockStore         \
        <u_fixSize + sizeof (ct_RefCount)> > { };

// Beispiel: BLOCK_STORE_DCL (ct_RndShort) expandiert zu
template <unsigned u_fixSize>
  class gct_RndShortBlockStore:
    public gct_BlockStore <ct_RndShortBlock, u_fixSize> { };
template <unsigned u_fixSize>
  class gct_RndShortBlockRefStore:
    public gct_RefStore <ct_RndShortBlockStore
      <u_fixSize + sizeof (ct_RefCount)> > { };
```

Das allgemeine Refstoretemplate gct_RefStore erwartet ähnlich wie das Blocktemplate als Parameter eine Storeklasse. Das Makro REF_STORE_DCL erzeugt einen vordefinierten Refstore und ähnelt dem Makro BLOCK_DCL. In REF_STORE_DCL (ct_StdChar) wird die Klasse ct_StdCharRefStore deklariert. Sie erweitert den globalen Store ct_StdCharStore um die Referenzzähler.

```
#define REF_STORE_DCL(StoreSpec)                  \
  class StoreSpec ## RefStore:                    \
    public gct_RefStore <StoreSpec ## Store> { };

// Beispiel: REF_STORE_DCL (ct_StdChar) expandiert zu
class ct_StdCharRefStore:
  public gct_RefStore <ct_StdCharStore> { };
```

Blöcke, Blockstores und Refstores werden etwa genauso häufig verwendet wie die globalen Stores. Deshalb generieren wir sie gemeinsam. Wir ergänzen das Makro GLOBAL_STORE_DCL um die drei Makroverwendungen BLOCK_DCL, BLOCK_STORE_DCL und REF_STORE_DCL. In diesem erweiterten Makro werden die folgenden Klassen und Templates generiert:

- eine globale Storeklasse,
- eine Blockklasse,
- ein Blockstoretemplate,
- ein Block-Refstoretemplate und
- eine globale Refstoreklasse.

```
#define GLOBAL_STORE_DCL(t_store, Obj, Size, t_size)               \
  class ct_ ## Obj ## Size ## Store                                \
    {                                                              \
    ....
    };                                                             \
  BLOCK_DCL (ct_ ## Obj ## Size)                                   \
  BLOCK_STORE_DCL (ct_ ## Obj ## Size)                             \
  REF_STORE_DCL (ct_ ## Obj ## Size)

// Beispiel: GLOBAL_STORE_DCL (ct_RndStore, Rnd, Int, unsigned int)
// expandiert zu
class ct_RndIntStore { ... };
class ct_RndIntBlock: public ....;
template <unsigned u_fixSize> gct_RndIntBlockStore: public ....;
template <unsigned u_fixSize> gct_RndIntBlockRefStore: public ....;
class ct_RndIntRefStore: public ....;
```

Unter der Deklaration der dynamischen Storeklasse `ct_StdStore` werden im Makro `GLOBAL_STORE_DCLS (ct_StdStore, Std)` die vordefinierten Instanzen der vier Größentypen `unsigned int`, `u. char`, `u. short` und `u. long` erzeugt. Abbildung 3-16 zeigt das Regal für Stores und Blöcke, die vom Standardstore gebildet werden. Ein ähnliches Regal existiert noch einmal für den Roundstore. Jede Box außer der ersten enthält vier Klassen oder Templates, von denen in der Abbildung die obersten beiden zu sehen sind. Sie unterscheiden sich jeweils durch ihren Größentyp.

```
class ct_StdStore
  {
  ....
  };

GLOBAL_STORE_DCLS (ct_StdStore, Std)
```

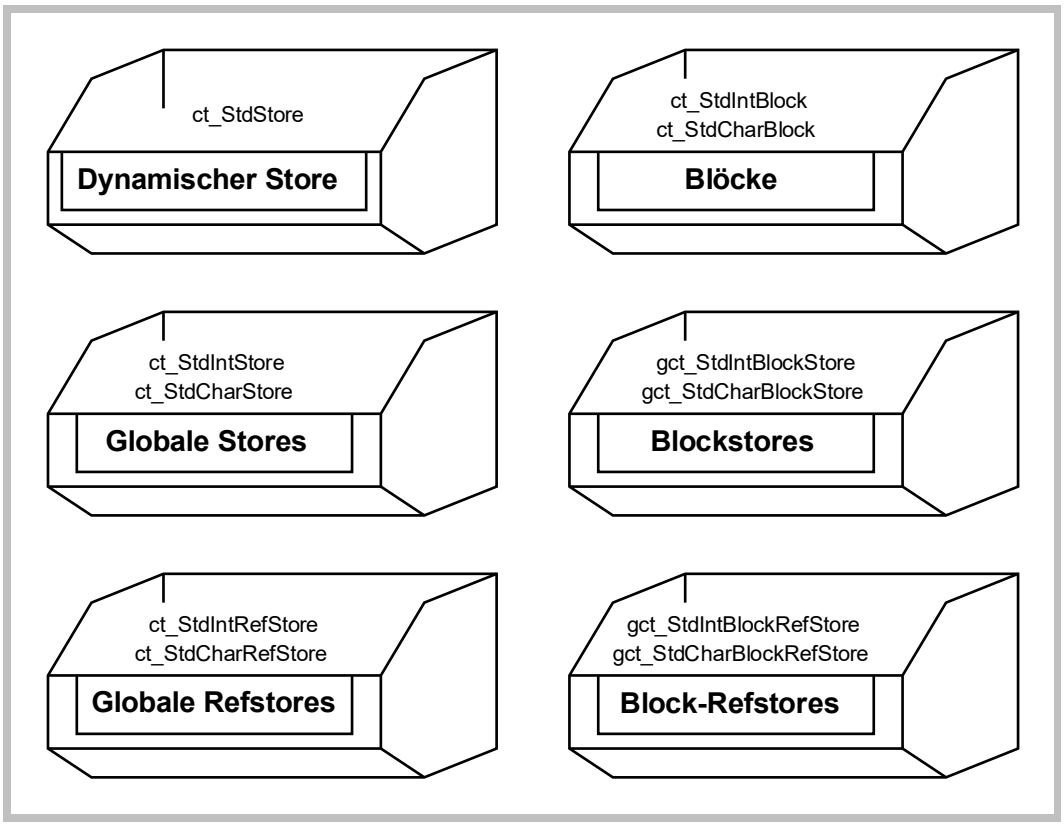

Abb. 3-16: *Regal für Stores und Blöcke*

3.4.2 Vordefinierte Strings und Container

Beim Generieren der Stringinstanzen stehen wir vor einem technischen Problem. Das Stringtemplate besitzt überladene Konstruktoren und Gleich-Operatoren. Diese werden nicht vererbt und stehen in einer abgeleiteten Klasse nicht zur Verfügung. Die im folgenden Programmfragment deklarierte Klasse verfügt über die vom Compiler generierten Methoden Standard-Konstruktor, Kopier-Konstruktor, Destruktor und Gleich-Operator. Der überladene Konstruktor und der Gleich-Operator mit dem Parametertyp `const char *` fehlen jedoch.

```
class ct_StdShortString: public gct_String <ct_StdShortBlock> { };
```

Eine Ausweichlösung wäre die Verwendung einer Typdefinition statt einer Klasse. Typdefinitionen werden aber vom Compiler an jeder Verwendungsstelle aufgelöst. Ist im Typ ein Template enthalten, verlangsamt sich das Übersetzen des Programms spürbar.

```
typedef gct_String <ct_StdShortBlock> t_StdShortString;
```

Eine befriedigende Lösung erhalten wir nur durch eine Klasse, in der die nicht vererbten Methoden neu definiert werden. Diese Methoden mappen die Funktionalität der Basisklasse. Sie können `inline` definiert werden und belasten nicht die Rechenzeit. Die Definition des Konstruktors und des Gleich-Operators mit dem Parametertyp `const char *` reicht nicht aus. Wurde ein eigener Konstruktor definiert, generiert der Compiler keine anderen Konstruktoren mehr. Wir müssen also sämtliche Konstruktoren und Gleich-Operatoren in die abgeleitete Klasse aufnehmen.

```
#define STRING_DCL(StoreSpec)                                    \
  class StoreSpec ## String:                                     \
    public gct_String <StoreSpec ## Block>                       \
```

```
    {                                                              \
  public:                                                          \
    inline StoreSpec ## String ();                                 \
    inline StoreSpec ## String (const char * pc_init);             \
    inline StoreSpec ## String (const StoreSpec ## String & co_init);  \
    inline StoreSpec ## String & operator = (const char * pc_asgn);    \
    inline StoreSpec ## String & operator =                        \
      (const StoreSpec ## String & co_asgn);                       \
    };                                                             \
  inline StoreSpec ## String:: StoreSpec ## String () { }          \
  inline StoreSpec ## String:: StoreSpec ## String                 \
    (const char * pc_init):                                        \
    gct_String <StoreSpec ## Block> (pc_init) { }                  \
  ....
  inline StoreSpec ## String & StoreSpec ## String::               \
    operator = (const StoreSpec ## String & co_asgn)               \
    {                                                              \
    gct_String <StoreSpec ## Block>:: operator = (co_asgn);        \
    return * this;                                                 \
    }

// Beispiel: STRING_DCL (ct_StdShort) expandiert zu
class ct_StdShortString:
  public gct_String <ct_StdShortBlock>
  {
public:
  inline ct_StdShortString ();
  ....
  };
....
inline ct_StdShortString & ct_StdShortString::
  operator = (const ct_StdShortString & co_asgn)
  {
  gct_String <ct_StdShortBlock>:: operator = (co_asgn);
  return * this;
  }
```

Im Makro `STRING_DCL` wird eine einzelne Stringklasse generiert. Zum Erzeugen der Klassen aller vier Größentypen nutzen wir das Makro `STRING_DCLS`. Es verwendet ähnlich wie `GLOBAL_STORE_DCLS` viermal das Makro für einen einzelnen Größentyp. Analoge Makros mit den Namen `ARRAY_DCLS` und `DLIST_DCLS` werden auch für die Container definiert.

```
#define STRING_DCLS(Obj)              \
  STRING_DCL (ct_ ## Obj ## Int)      \
  STRING_DCL (ct_ ## Obj ## Char)     \
  STRING_DCL (ct_ ## Obj ## Short)    \
  STRING_DCL (ct_ ## Obj ## Long)

// Beispiel: STRING_DCLS (Rnd) expandiert zu
class ct_RndIntString: public ....;
class ct_RndCharString: public ....;
class ct_RndShortString: public ....;
class ct_RndLongString: public ....;
```

Das Arraytemplate besitzt zwei Parameter, die Objektklasse `t_obj` und die Blockklasse `t_block`. Der erste Parameter bleibt auch bei vordefinierten Instanzen erhalten. In `ARRAY_DCL (ct_RndLong)` wird das Template `gct_RndLongArray` deklariert. Es baut auf der generierten Blockklasse `ct_RndLongBlock` auf.

```
#define ARRAY_DCL(StoreSpec)                          \
  template <class t_obj>                              \
    class g ## StoreSpec ## Array:                    \
      public gct_Array <t_obj, StoreSpec ## Block> { };
```

```
// Beispiel: ARRAY_DCL (ct_RndLong) expandiert zu
template <class t_obj>
  class gct_RndLongArray:
    public gct_Array <t_obj, ct_RndLongBlock> { };
```

Das Template `gct_DList` erwartet als zweiten Parameter eine Storeklasse. Dabei ist die Auswahl größer als bei Blockklassen (siehe Abbildung 3-16). Für einen einzelnen Größentyp generieren wir vier Listentemplates: Eine normale Liste, eine Blockliste, eine Refliste und eine Block-Refliste. Im Makro `DLIST_DCL` werden vordefinierte Stores verwendet. Zum Beispiel nutzt der Listencontainer `gct_RndIntBlockDList` das Storetemplate `gct_RndIntBlockStore`.

```
#define DLIST_DCL(StoreSpec)                                          \
  template <class t_obj>                                              \
    class g ## StoreSpec ## DList:                                    \
      public gct_DList <t_obj, StoreSpec ## Store> { };               \
  template <class t_obj>                                              \
    class g ## StoreSpec ## BlockDList:                               \
      public gct_DList <t_obj, g ## StoreSpec ## BlockStore           \
      <sizeof (gct_DListNode <t_obj, StoreSpec ## Store:: t_Size>)> >{};\
  template <class t_obj>                                              \
    class g ## StoreSpec ## RefDList:                                 \
      public gct_RefDList <t_obj, StoreSpec ## RefStore> { };         \
  template <class t_obj>                                              \
    class g ## StoreSpec ## BlockRefDList:                            \
      public gct_RefDList <t_obj, g ## StoreSpec ## BlockRefStore     \
      <sizeof (gct_DListNode <t_obj, StoreSpec ## Store:: t_Size>)> >{};

// Beispiel: DLIST_DCL (ct_RndInt) expandiert zu
template <class t_obj>
  class gct_RndIntDList:
    public gct_DList <t_obj, ct_RndIntStore> { };
template <class t_obj>
  class gct_RndIntBlockDList:
    public gct_DList <t_obj, gct_RndIntBlockStore
      <sizeof (gct_DListNode <t_obj, ct_RndIntStore:: t_Size>)> > { };
template <class t_obj>
  class gct_RndIntRefDList:
    public gct_RefDList <t_obj, ct_RndIntRefStore> { };
template <class t_obj>
  class gct_RndIntBlockRefDList:
    public gct_RefDList <t_obj, gct_RndIntBlockRefStore
      <sizeof (gct_DListNode <t_obj, ct_RndIntStore:: t_Size>)> > { };
```

In Abbildung 3-17 sehen wir das Regal für Strings und Container, die vom Roundstore gebildet werden. Zu ihrer Erzeugung sind die drei Makros `STRING_DCLS (Rnd)`, `ARRAY_DCLS (Rnd)` und `DLIST_DCLS (Rnd)` erforderlich. In jeder Box des Regals befinden sich vier Klassen oder Templates, von denen die obersten beiden zu sehen sind. Ein ähnliches Regal existiert auch für den Standardstore.

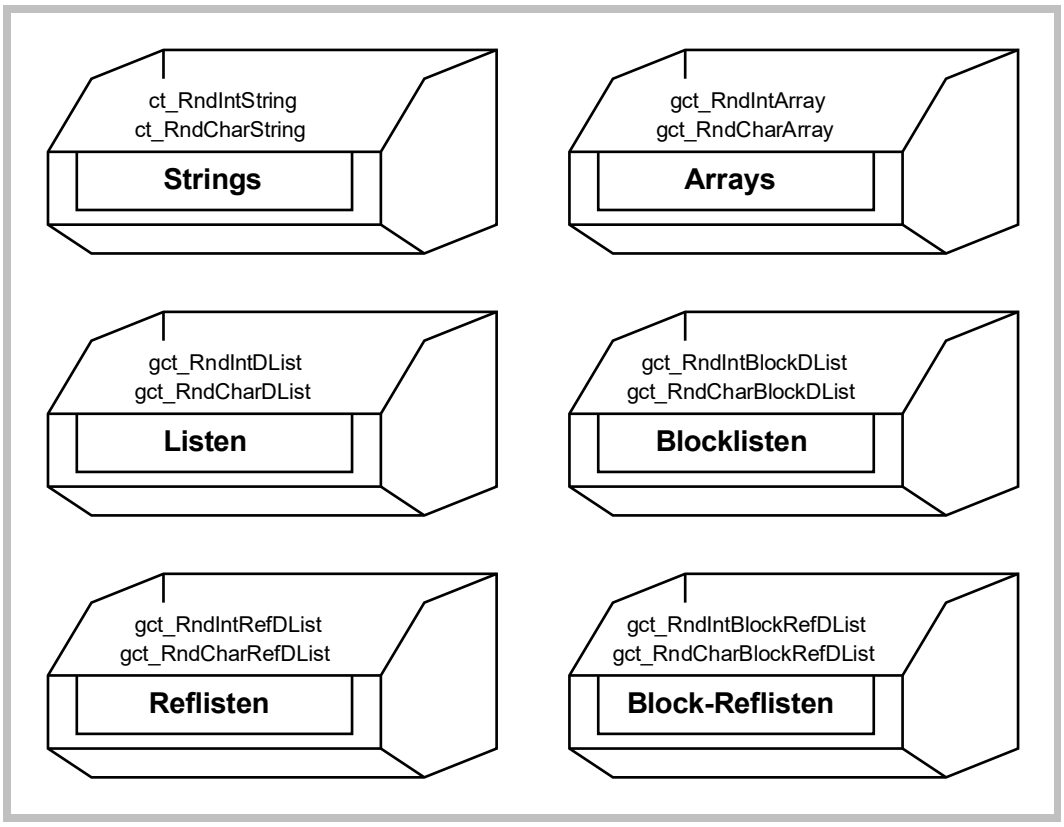

Abb. 3-17: *Regal für Strings und Container*

3.4.3 Collections

Mit großen Regalen voller Arrays und Listen fällt es uns nicht schwer, einige Collections zu implementieren. Wir benötigen dazu ein Template, das die Funktionalität eines Containers auf das Collectioninterface (siehe Abschnitt 2.4.2) mappt. Das Template `gct_Collection` erwartet eine Containerklasse der Form `gct_AnyContainer <ct_Object *>`, also einen Container, der C++-Zeiger auf die abstrakte Basisklasse `ct_Object` enthält. Beim Zugriff auf ein Objekt ist ein Dereferenzieren erforderlich. Die Containermethode `GetObj` liefert einen Zeiger auf einen Zeiger (`ct_Object * *`). Die Collectionmethode `GetObj` muß aber einen Zeiger auf ein `ct_Object` zurückgeben. Umgekehrt muß beim Einfügen eines Zeigers mit der Collectionmethode `AddPtrAfter` die Adresse des Zeigers an die Containermethode `AddObjAfter` gegeben werden.

```
template <class t_cont>
  class gct_Collection: public ct_Collection
    {
    t_cont                      o_Container;
  public:
    virtual inline const char * GetTypeName () const;
    virtual inline t_CollLen    GetLen () const;
    virtual inline t_CollPtr    First () const;
    virtual inline t_CollPtr    Next (t_CollPtr o_ptr) const;
    virtual inline ct_Object *  GetObj (t_CollPtr o_ptr) const;
    virtual inline t_CollPtr    AddPtr (ct_Object * pco_obj);
    virtual inline t_CollPtr    AddPtrCond (ct_Object * pco_obj);
    virtual inline t_CollPtr    AddPtrAfter (t_CollPtr o_ptr,
                                  ct_Object * pco_obj);
    virtual inline t_CollPtr    DelPtr (t_CollPtr o_ptr);
    };
```

```
template <class t_cont>
  inline ct_Object *
  gct_Collection <t_cont>:: GetObj (t_CollPtr o_ptr) const
    {
    return * o_Container. GetObj ((t_cont:: t_Pointer) o_ptr);
    }

template <class t_cont>
  inline t_CollPtr gct_Collection <t_cont>:: AddPtrAfter
  (t_CollPtr o_ptr, ct_Object * pco_obj)
    {
    return (t_CollPtr)
      o_Container. AddObjAfter ((t_cont:: t_Pointer) o_ptr, & pco_obj);
    }
```

Erzeugen wir mit diesem Template eine konkrete Collection, müssen wir nur noch die Methode `GetTypeName` definieren. Bei der Auswahl eines geeigneten Containers bevorzugen wir die vordefinierten Instanzen des Roundstores. Diese arbeiten auf der Grundlage einer effektiveren Speicherverwaltung. Der globale Größentyp für Collections `t_CollLen` ist auf `unsigned long` definiert. Für eine optimale Typverträglichkeit der Collection mit ihrer Basisklasse verwenden wir Container mit dem geschachtelten Größentyp `unsigned long`, zum Beispiel `gct_RndLongArray`.

```
class ct_Array: public gct_Collection <gct_RndLongArray <ct_Object *> >
  {
public:
  virtual inline const char * GetTypeName () const;
  };

inline const char * ct_Array:: GetTypeName () const
  {
  return "ct_Array";
  }
```

Die große Auswahl an Listencontainern ermöglicht es uns, mehrere Listencollections zu implementieren. Die neue Klasse `ct_DList` baut auf einem normalen Listencontainer auf und besitzt dasselbe Verhalten wie die gleichnamige Klasse unseres Beispielprogramms OHelp (siehe Abschnitt 1.4.2). Eine bessere Speicherauslastung besitzt die Klasse `ct_BlockDList`. Sie wird mit Hilfe des Containertemplates `gct_RndLongBlockDList` implementiert.

```
class ct_DList: public gct_Collection <gct_RndLongDList <ct_Object *> >
  {
public:
  virtual inline const char * GetTypeName () const;
  };

class ct_BlockDList:
  public gct_Collection <gct_RndLongBlockDList <ct_Object *> >
  {
public:
  virtual inline const char * GetTypeName () const;
  };
```

Wollen wir die Funktionalität der Reflisten auf die Collections übertragen, benötigen wir eine abstrakte Basisklasse. Diese erweitert das allgemeine Collectioninterface um die Zugriffsmethoden auf den Referenzzähler, der jedem Element zugeordnet wird. Die abstrakte Klasse `ct_RefCollection` dient der Verarbeitung von Collections mit Referenzzählern in einem polymorphen Kontext. Zum Mappen der Funktionalität eines Refcontainers auf eine Refcollection nutzten wir das Template `gct_RefCollection`.

```
class ct_RefCollection: public ct_Collection
```

```
  {
public:
  virtual void         IncRef (t_CollPtr o_ptr) = 0;
  virtual void         DecRef (t_CollPtr o_ptr) = 0;
  virtual t_RefCount   GetRef (t_CollPtr o_ptr) = 0;
  virtual bool         IsAlloc (t_CollPtr o_ptr) = 0;
  virtual bool         IsFree (t_CollPtr o_ptr) = 0;
  };

template <class t_cont>
  class gct_RefCollection: public ct_RefCollection
    {
    t_cont                      o_Container;
  public:
    virtual inline const char * GetTypeName () const;
    virtual inline t_CollLen    GetLen () const;
    .... // Weitere Collectionmethoden
    virtual inline void         IncRef (t_CollPtr o_ptr);
    .... // Weitere Zugriffsmethoden auf den Referenzzähler
    };

template <class t_cont>
  inline bool gct_RefCollection <t_cont>:: IsAlloc (t_CollPtr o_ptr)
    {
    return o_Container. IsAlloc ((t_cont:: t_Pointer) o_ptr);
    }
```

Mit diesem Template können wir konkrete Refcollections implementieren. Als Grundlage nutzen wir zwei Refcontainer, eine normale Refliste und eine Block-Refliste. Die beiden neuen Collections nennen wir `ct_RefDList` und `ct_BlockRefDList`. In Abbildung 3-18 sehen wir die verwendeten Klassen der Collection `ct_RefDList`. Der zugrundegelegte Refcontainer und seine Abhängigkeiten sind in der Abbildung nicht enthalten.

```
class ct_RefDList:
  public gct_RefCollection <gct_RndLongRefDList <ct_Object *> >
  {
public:
  virtual inline const char * GetTypeName () const;
  };

class ct_BlockRefDList:
  public gct_RefCollection <gct_RndLongBlockRefDList <ct_Object *> >
  {
public:
  virtual inline const char * GetTypeName () const;
  };
```

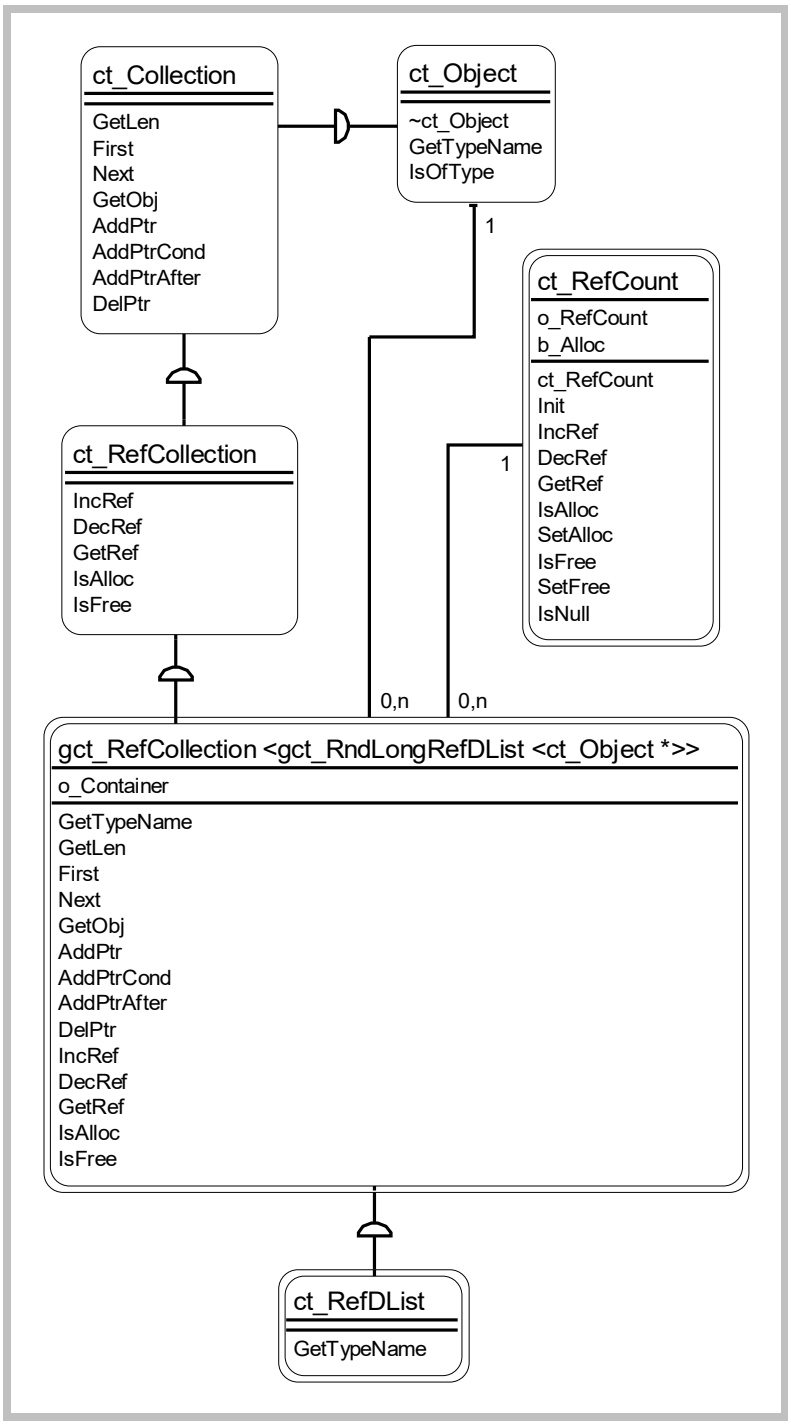

Abb. 3-18: *Verwendete Klassen einer Refcollection*

Die vordefinierten Collections bieten elementare Möglichkeiten zur Anpassung an konkrete Bedingungen. Zur Auswahl stehen eine Array- und vier Listencollections. Es können jedoch auch weitergehende Wünsche auftreten. Zum Beispiel kann beim vordefinierten Roundstore der Schritt-Teiler Vier eingestellt sein. Wir möchten aber für eine Blockcollection die binäre Rundung (Schritt-Teiler Eins) verwenden. Dazu müssen wir einen eigenen Roundstore und eine eigene Collection definieren. Die Makros `GLOBAL_STORE_DCLS` und `DLIST_DCLS` nehmen uns den größten Teil der Arbeit ab. Im folgenden Programmfragment wird das globale Roundstoreobjekt `co_MyCollStore` erzeugt. Am Programmbeginn wird die binäre Rundung eingestellt. Die Collection `ct_MyBlockDList` fordert ihren Speicher von diesem globalen Store an.

```
// In einer Headerdatei plazieren
```

```
GLOBAL_STORE_DCLS (ct_RndStore, MyColl)
DLIST_DCLS (MyColl)

class ct_MyBlockDList:
  public gct_Collection <gct_MyCollLongBlockDList <ct_Object *> >
  {
public:
  virtual inline const char * GetTypeName () const;
  };

inline const char * ct_MyBlockDList:: GetTypeName () const
  {
  return "ct_MyBlockDList";
  }

// In einer Implementierungsdatei plazieren
GLOBAL_STORE_DEFS (ct_RndStore, MyColl)

int main ()
  {
  GetMyCollStore ()-> SetMinSize (16);
  GetMyCollStore ()-> SetStepDiv (1);
  ct_MyBlockDList co_myList;
  ....
  }
```

3.5 OHelp2

3.5.1 Implementierung

Am Ende des dritten Teils des Buches werden wir die neuen Konzepte anhand unseres Beispiels OHelp überprüfen. Das Design übernehmen wir aus dem ersten Teil (siehe Abschnitt 1.5.1). Bei der Implementierung setzen wir jedoch die neuen Programmbausteine ein und nennen das neue Beispiel OHelp2.

Als Grundlage der Speicherverwaltung verwenden wir den globalen Roundstore. Er vergrößert zwar jeden dynamischen Speicherblock zum Unterbringen der gerundeten Größe. Durch die Rundung bleibt aber die Freikette kleiner, und die Speicherverwaltung wird insgesamt schneller. Das globale Objekt `co_RndStore` taucht in unserem Programm nirgendwo direkt auf. In den vordefinierten Instanzen (zum Beispiel `gct_RndShortRefDList`) ist aber sein Name enthalten.

In den Anwendungsklassen von OHelp2 können wir überall den geschachtelten Größen- bzw. Längentyp `unsigned short` einsetzen. Zum Beispiel ist es sinnvoll, Zeichenketten auf eine Größe von 64 KB zu beschränken. Die Klasse `ct_RndShortString` umfaßt nur sechs Bytes, während die Klasse `ct_RndIntString` acht Bytes benötigt. Zum Speichern der Hyperlinks und Formate in einem Thema verwenden wir den Arraycontainer `gct_RndShortArray`. Der dynamische Block dieses Arrays kann bis zu 64 KB groß werden. Auch diese Beschränkung ist in unserer Anwendung sinnvoll. Das Arrayobjekt selbst umfaßt nur sechs Bytes.

Datentypen, die wir häufig verwenden, kürzen wir mit einer Typdefinition ab. Zum Speichern des Texts in einem Thema nutzen wir die Klasse `ct_RndShortString`. Positionen, die sich auf diesen Text beziehen, sind in Hyperlinks und Formatangaben enthalten. Der Datentyp für Textpositionen ist gleich dem geschachtelten Größentyp der Stringklasse. Beide Datentypen sollten im Programmtext hintereinander stehen. Ändern wir später die Stringklasse, muß auch der Datentyp für Textpositionen überprüft werden.

```
typedef ct_RndShortString t_TextString;
typedef t_TextString:: t_Size t_TextSize; // unsigned short
```

Analog verfahren wir bei Containern. Ein Hyperlink enthält einen Verweis auf ein Thema. Dazu verwenden wir keinen C++-Zeiger, sondern den logischen Zeiger aus der Liste aller Themen. Den Containertyp für Themen können wir an dieser Stelle nicht definieren. Die Klasse `ct_Topic` ist dem Compiler noch unbekannt. Eine Vorwärtsdeklaration reicht in diesem Fall nicht aus, denn der Container enthält Objekte vom Typ `ct_Topic` (keine Zeiger). Damit beide Datentypen textuell beisammen stehen, definieren wir den Containertyp als ein Präprozessormakro.

In OHelp1 besaß jedes Thema einen Referenzzähler. Ein Hyperlink, das auf dieses Thema verwies, erhöhte ihn um eins. Ein Thema durfte erst gelöscht werden, wenn keine Verweise mehr darauf existierten. Nun verwenden wir zum Sammeln der Themen eine Refliste, also eine Liste, die jedem Eintrag einen Referenzzähler zuweist. Wir können fragen, ob ein Hyperlink noch gültig ist. Ein Thema kann unabhängig vom zugeordneten Referenzzähler gelöscht werden.

```
#define TOPIC_LIST gct_RndShortRefDList <ct_Topic>
typedef void * t_TopicPtr;
```

Die Klassen `ct_HyperLink` und `ct_Format` müssen nicht mehr von der abstrakten Basisklasse `ct_Object` erben. Aber sie müssen einige neue Anforderungen erfüllen. Das Einfügen in einen Container erfolgt mit dem Standard- oder Kopier-Konstruktor. Andere Konstruktoren können nicht verwendet werden. Die Konstruktoren aus dem Beispiel OHelp1 werden deshalb in normale Methoden umgewandelt. Ein Objekt wird nun mit seinem Standard-Konstruktor erzeugt und mit einer `Init`-Methode initialisiert. Für die Containermethode `AddObjCond` muß die Klasse einen Gleichheits-Operator enthalten.

```
class ct_HyperLink
  {
  t_TextSize           o_Pos;
  t_TopicPtr           o_TopicPtr;
  bool                 b_InText: 1;
public:
  inline void          Init (t_TopicPtr o_ptr);
  inline void          Init (t_TextSize o_pos, t_TopicPtr o_ptr);
  inline bool          operator == (const ct_HyperLink & co_comp) const;
  inline t_TextSize    GetPos () const;
  inline void          MovePos (int i_delta);
  inline t_TopicPtr    GetTopicPtr () const;
  inline bool          IsInText () const;
  };

typedef unsigned char t_FormatId;

const t_FormatId co_Bold      = 0x01;
const t_FormatId co_Italic    = 0x02;
const t_FormatId co_Underline = 0x04;
const t_FormatId co_Example   = 0x08;

class ct_Format
  {
  t_TextSize           o_Pos;
  t_TextSize           o_Len;
  t_FormatId           o_Format;
public:
  inline void          Init (t_TextSize o_pos, t_TextSize o_len,
                         t_FormatId o_format);
  inline bool          operator == (const ct_Format & co_comp) const;
```

```
  inline t_TextSize    GetPos () const;
  inline void          MovePos (int i_delta);
  inline t_TextSize    GetLen () const;
  inline void          ChangeLen (int i_delta);
  inline t_FormatId    GetFormat () const;
  inline void          AddFormat (t_FormatId o_format);
  inline void          DelFormat (t_FormatId o_format);
  };
```

Zum Speichern der Hyperlinks und Formate verwenden wir das Template `gct_RndShortArray`. Die konkreten Containerklassen werden nicht mit einer Typdefinition erzeugt. Zum Beschleunigen des Compilierens definieren wir dafür zwei Klassen. Die zugeordneten Zeigertypen sind primitive Datentypen und können als Typdefinition notiert werden. In OHelp1 war für den Zugriff auf ein Hyperlink-Objekt die Methode `ct_Topic:: GetHyperLink` nötig. Sie ermittelte aus der Collection einen Zeiger des Typs `ct_Object *` und wandelte ihn in `ct_HyperLink *` um. Diese Typumwandlung ist nun nicht mehr erforderlich. Beim Durchlaufen des Containers erhalten wir bereits die richtigen Zeiger.

```
class ct_HyperLinks: public gct_RndShortArray <ct_HyperLink> { };
typedef ct_HyperLinks:: t_Pointer t_HyperLinkPtr; // unsigned short

class ct_Formats: public gct_RndShortArray <ct_Format> { };
typedef ct_Formats:: t_Pointer t_FormatPtr; // unsigned short
```

In der Klasse `ct_Topic` sind nur geringe Änderungen nötig. Das Attribut `u_RefCount` entfällt, denn in der Liste aller Themen ist jedem Eintrag ein Referenzzähler zugeordnet. Stattdessen benötigen wir den logischen Zeiger des Themas innerhalb der Themenliste. Er gelangt als Attribut `o_LogPtr` neu in die Klasse. Der logische Zeiger wird zusammen mit dem Zeiger auf den Hypertext und dem Namen in der `Init`-Methode initialisiert und kann später nicht mehr geändert werden. Er wird in der Methode `GetRefCount` verwendet (siehe unten).

Ein einzelnes Hyperlink-Objekt kann keine Auskunft geben, ob es noch gültig ist, denn es besitzt keinen Zugriff auf die Liste aller Themen. Die Methode `IsHyperLinkValid` wird also in der Klasse `ct_Topic` deklariert. Am Ende des folgenden Programmausschnitts sehen wir ihre Definition. Sie erwartet als Parameter den logischen Zeiger des Hyperlinks. Aus dem Hyperlink-Container `co_HyperLinks` ermittelt sie den logischen Zeiger des referenzierten Themas. Damit wird auf die Themenliste zugegriffen. Die Reflisten-Methode `IsAlloc` liefert schließlich die gewünschte Information.

```
class ct_HyperText;

class ct_Topic
  {
  ct_HyperText *       pco_HyperText;
  t_TopicPtr           o_LogPtr;
  ct_RndShortString    co_Title;
  t_TextString         o_Text;
  ct_HyperLinks        co_HyperLinks;
  ct_Formats           co_Formats;

  void                 Clear ();
  void                 Copy (const ct_Topic & co_copy);
public:
                       ct_Topic ();
                       ct_Topic (const ct_Topic & co_init);
                       ~ct_Topic ();
  void                 Init (ct_HyperText * pco_hyperText,
                         t_TopicPtr o_ptr, const char * pc_title);
  ct_Topic &           operator = (const ct_Topic & co_asgn);
  bool                 operator == (const ct_Topic & co_comp) const;
  inline ct_HyperText * GetHyperText () const;
```

```
  inline t_TopicPtr    GetLogPtr () const;
  unsigned             GetRefCount () const;
  inline const char *  GetTitle () const;
  inline void          SetTitle (const char * pc_title);
  inline const char *  GetText () const;
  void                 InsertText (t_TextSize o_pos, const char * pc_tx);
  void                 DeleteText (t_TextSize o_pos, t_TextSize o_len);
  inline const ct_HyperLinks * GetHyperLinks () const;
  bool                 IsHyperLinkValid (t_HyperLinkPtr o_ptr) const;
  t_HyperLinkPtr       AddHyperLink (t_TopicPtr o_ptr);
  t_HyperLinkPtr       AddHyperLink (t_TextSize o_pos, t_TopicPtr o_ptr);
  t_HyperLinkPtr       DelHyperLink (t_HyperLinkPtr o_ptr);
  inline const ct_Formats * GetFormats () const;
  t_FormatPtr          AddFormat (t_TextSize o_pos, t_TextSize o_len,
                         t_FormatId o_format);
  t_FormatPtr          DelFormat (t_FormatPtr o_ptr);
  };

unsigned ct_Topic:: GetRefCount () const
  {
  return pco_HyperText-> GetTopics ()-> GetRef (o_LogPtr);
  }

bool ct_Topic:: IsHyperLinkValid (t_HyperLinkPtr o_ptr) const
  {
  return pco_HyperText-> GetTopics ()->
    IsAlloc (co_HyperLinks. GetObj (o_ptr)-> GetTopicPtr ());
  }
```

Die Klasse `ct_HyperText` können wir im wesentlichen von OHelp1 übernehmen. Wir ändern nur die Typen der Attribute und Methodenparameter. Die Methode `GetTopic` für den Zugriff auf ein einzelnes Thema entfällt. Der Listentyp für die Themen wird mit Hilfe des weiter oben definierten Makros erzeugt. Wir können keine platzsparende Blockliste einsetzen, denn ein Hypertext soll bis zu 10 000 Themen aufnehmen können. Damit ist eine Blockliste überfordert.

```
class ct_Topics: public TOPIC_LIST { };

class ct_HyperText
  {
  ct_RndShortString    co_Name;
  t_TopicPtr           o_RootTopicPtr;
  ct_Topics            co_Topics;
public:
                       ct_HyperText ();
                       ~ct_HyperText ();
  inline const char *  GetName () const;
  inline void          SetName (const char * pc_name);
  inline t_TopicPtr    GetRootTopicPtr () const;
  inline void          SetRootTopicPtr (t_TopicPtr o_rootPtr);
  inline const ct_Topics * GetTopics () const;
  t_TopicPtr           AddTopic (const char * pc_title);
  t_TopicPtr           CopyTopic (t_TopicPtr o_source,
                         const char * pc_newTitle);
  void                 ReplaceTopic (t_TopicPtr o_repl, t_TopicPtr o_src,
                         const char * pc_newTitle);
  void                 DelTopicUsages (t_TopicPtr o_ptr);
  t_TopicPtr           DelTopic (t_TopicPtr o_ptr);
  };
```

Der Verzicht auf abstrakte Basisklassen und virtuelle Methoden bei der Implementierung von OHelp2 ist *keine allgemeine Empfehlung*. Unsere Programmierregel aus dem Abschnitt 1.6.1 lautet: An *performancekritischen Stellen* suchen wir eine Lösung ohne virtuelle Methoden.

Existiert diese Lösung nicht, setzen wir weiterhin virtuelle Methoden ein. In unserem Beispiel OHelp2 existieren jedoch solche Lösungen.

3.5.2 Performance-Analyse

Die neue Implementierung von OHelp unterziehen wir demselben kritischen Test wir OHelp1. Zunächst starten wir ein Testprogramm. Dann ermitteln wir rechnerisch den Ressourcenverbrauch von OHelp2. Wir nutzen dazu dieselben Daten wie beim ersten Performancetest. Zur Erinnerung seien diese Zahlen noch einmal wiederholt.

- 10 000 Themen,
- Titel des Themas mit 15 Zeichen,
- pro Thema 10 Zeilen,
- pro Zeile 30 Zeichen Text und
- pro Zeile je eine Formatangabe und ein Hyperlink.

Im Testprogramm wird zuerst die algorithmische Richtigkeit anhand kleiner Beipiele überprüft. Dann erfolgt der Performancetest. Wir bauen eine größere Datenmenge auf und nehmen daran umfangreiche Änderungen vor. Der Test verläuft diesmal ohne Probleme. Unser Programm ist merklich schneller geworden und benötigt weniger Speicher. Auch nach längerer Arbeit mit großen Datenmengen wird das Programm nicht langsamer. Die höhere Rechengeschwindigkeit ist auf folgende Faktoren zurückzuführen:

- Kleine Methoden sind `inline` definiert und können an jeder Verwendungsstelle `inline` expandiert werden.
- An rechenzeitkritischen Stellen werden keine virtuellen Methoden eingesetzt.
- Die Speicherverwaltung wird durch Rundung und minimierte Blockanzahl entlastet.

Zur Berechnung der absoluten Größe von Objekten setzen wir wieder einen 32-Bit-Compiler voraus. Die primitiven Datentypen umfassen: `char` ein Byte, `short` zwei Bytes, `int` vier Bytes, `long` vier Bytes und Zeiger vier Bytes. Einzelne Objekte der Typen `ct_HyperLink`, `ct_Format` und `ct_Topic` beanspruchen keine eigenen Speicherblöcke, denn sie werden in Containern untergebracht. In Abbildung 3-19 sehen wir das Speicherlayout eines einzelnen Themas. Tabelle 3-6 enthält die Analyseergebnisse.

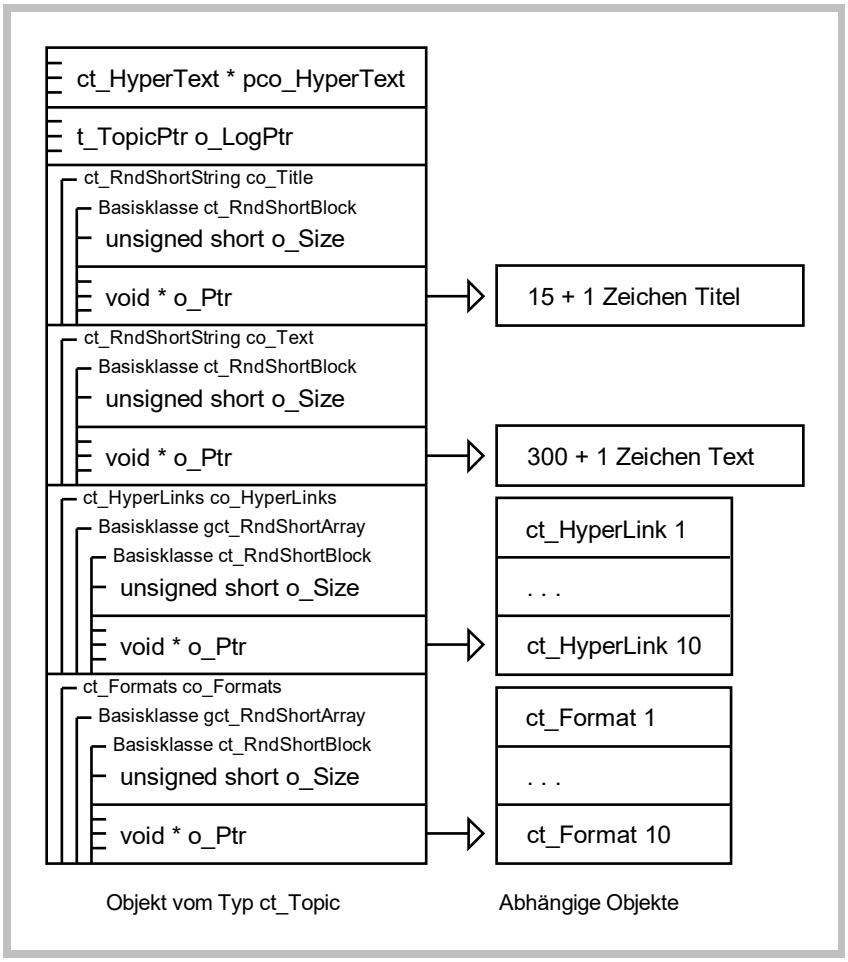

Abb. 3-19: *Speicherlayout einer Instanz der Klasse ct_Topic*

Objekttyp	Absolute Größe	Virt. Tab.-Zeiger	Anzahl Blöcke
Hyperlink	7 Bytes	0 Bytes	0
Formatangabe	5 Bytes	0 Bytes	0
Thema ohne Inhalt	32 Bytes	0 Bytes	0
Titel mit 15 Zeichen	15 + 1 Bytes	0 Bytes	1
Text mit 300 Zeichen	300 + 1 Bytes	0 Bytes	1
10 Hyperlinks	70 Bytes	0 Bytes	1
10 Formatangaben	50 Bytes	0 Bytes	1
Themen-Inhalt	437 Bytes	0 Bytes	4
Thema mit Inhalt (OHelp2)	469 Bytes	0 Bytes	4
Thema mit Inhalt (OHelp1)	777 Bytes	100 Bytes	25

Tab. 3-6: *Speicheranalyse der Klasse ct_Topic*

Bei der Auswertung der Tabelle müssen wir beachten, daß in einem Thema 317 Bytes reine Zeichenketten enthalten sind, die sich nicht optimieren lassen. Dennoch konnte der Speicherbedarf deutlich verringert werden. Durch den Einsatz von Containern wurde auch die Anzahl der Blöcke stark reduziert. Damit sinkt der unsichtbare Verwaltungsaufwand der

dynamischen Speicherverwaltung. Abbildung 3-20 zeigt das Speicherlayout eines Hypertexts. In Tabelle 3-7 befinden sich die Ergebnisse der Speicheranalyse.

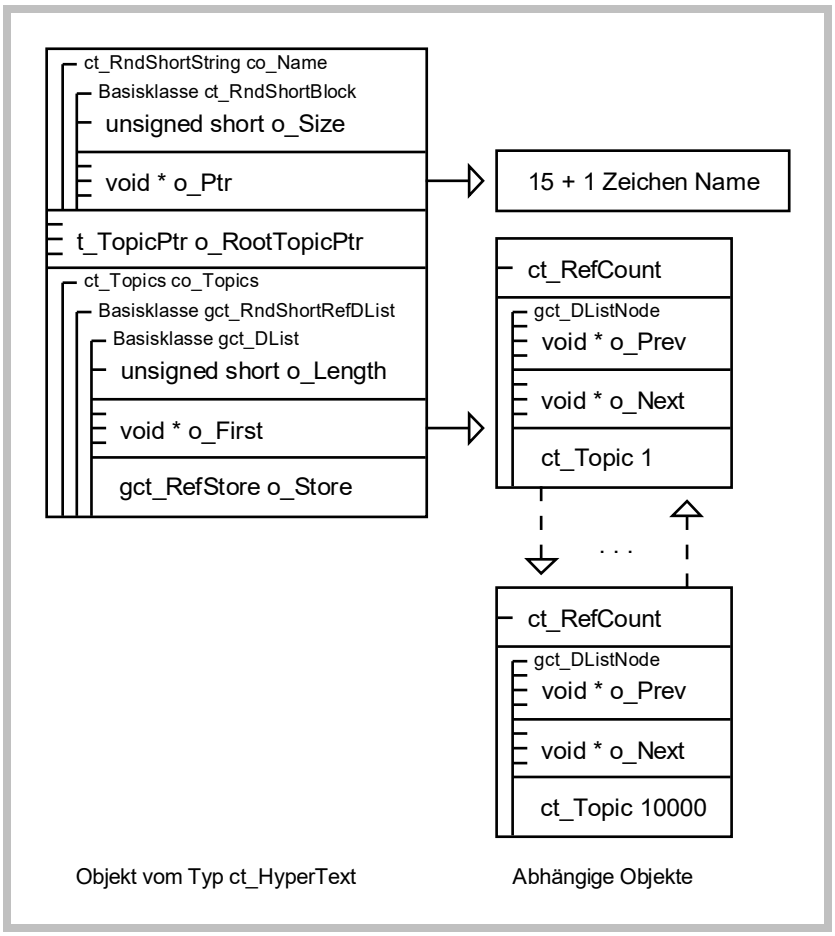

Abb. 3-20: *Speicherlayout einer Instanz der Klasse ct_HyperText*

Objekttyp	Absolute Größe	Virt. Tab.-Zeiger	Anzahl Blöcke
Hypertext ohne Inhalt	17 Bytes	0 Bytes	1
Titel mit 15 Zeichen	15 + 1 Bytes	0 Bytes	1
DList-Eintrag mit Th. o. I.	42 Bytes	0 Bytes	1
10 000 DList-Einträge	42 000 Bytes	0 Bytes	10 000
Hypertext mit Th. o. I.	42 033 Bytes	0 Bytes	10 002
10 000 Themen-Inhalte	4 370 000 Bytes	0 Bytes	40 000
Hypertext mit Inhalt (OHelp2)	4 412 033 Bytes	0 Bytes	50 002
Hypertext mit Inhalt (OHelp1)	7 890 048 Bytes	1 000 012 Bytes	260 002

Tab. 3-7: *Speicheranalyse der Klasse ct_HyperText*

Ein Vergleich mit den Analyseergebnissen von OHelp1 ergibt, daß wir den Speicherbedarf auf 56 und die Anzahl der Blöcke auf 19 Prozent reduziert haben. Die wichtigsten Faktoren unserer Speicheroptimierung sind:

- Wir setzen angepaßte primitive Datentypen ein, zum Beispiel `unsigned short` statt `unsigned int`.
- Unsere Daten enthalten keine überflüssigen virtuellen Tabellenzeiger.
- Container enthalten keine Zeiger zum Verwalten der Objekte.

Bei der Neuimplementierung von OHelp blieb das Interface der Klassen weitgehend stabil. Die Benutzeroberfläche passen wir in kürzester Zeit an den neuen Programmkern an. Das einst so schwerfällige Programm wirkt nun flott und elegant. Die gestreßten Anwender unseres Hilfesystems werden umgehend telefonisch benachrichtigt. Schon nächste Woche erhalten sie eine neue Version, die doppelt so schnell ist und nur noch halb so viel Speicher verbraucht.