Xpert.press

Die Reihe **Xpert.press** vermittelt Professionals in den Bereichen Softwareentwicklung, Internettechnologie und IT-Management aktuell und kompetent relevantes Fachwissen über Technologien und Produkte zur Entwicklung und Anwendung moderner Informationstechnologien.

Ralf Schneeweiß

Moderne C++ Programmierung

Klassen, Templates, Design Patterns

Mit 19 Abbildungen und 397 Listings

Springer

Ralf Schneeweiß
Gölzstraße 8
72072 Tübingen
ralf.schneeweiss@oop-trainer.de

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

ISSN 1439-5428
ISBN-10 3-540-22281-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-22281-1 Springer Berlin Heidelberg New York

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

Satz: ptp, Berlin
Herstellung: LE-TEX, Jelonek, Schmidt & Vöckler GbR, Leipzig
Umschlaggestaltung: KünkelLopka Werbeagentur, Heidelberg
Gedruckt auf säurefreiem Papier 33/3142 YL – 5 4 3 2 1 0

Vorwort

Um im Jahr 2006 noch ein Buch über C++ zu schreiben, bedurfte es meinerseits einiger Vorüberlegungen, bis ich mich zu diesem Schritt entscheiden konnte. Da ist zum einen die aktuelle Veränderung in der Nutzung von Programmiersprachen in der Anwendungsentwicklung, die ein deutliches Zeichen für die Zukunft von Java und anderen moderneren Programmiersprachen ist und ein ebensolches Zeichen für das allmähliche Verschwinden von C++ aus dieser Domäne. Zum anderen ist da ein scheinbares Defizit in der Sprache C++ selbst, die sich ein Stück weit gegen die unmittelbare Anwendung neuer Technologietrends zu sträuben scheint. Die enorme Komplexität der Sprache ist dafür sicher ein Grund unter anderen. Was sind also meine Gründe, doch ein Buch über diese alte und noch in vielen Bereichen unersetzliche Sprache zu schreiben?

In den letzten Jahren fand eine rasante Entwicklung bei den Mobile Devices, den Embedded Systems und den vermittelnden Systemen in Netzen statt. Diese Entwicklung dauert auch heute noch an. Es entsteht viel mehr neue Hardware als in den neunziger Jahren und es entsteht die dazugehörige Betriebssystemlandschaft. Gerade für diese neuen Systeme braucht man heute C++ als systemnahe und deterministische Programmiersprache. Da heute aufgrund der besseren Portierbarkeit mancher gängigen Compiler C++ auch auf fast allen neuen Plattformen zur Verfügung steht, erfährt die Sprache genau dort eine späte Verbreitung. Häufig werden Systeme und Teile davon in Crosscompileumgebungen entwickelt, die Windows oder Linux als Entwicklungsbasis definieren und ein Embedded System als Zielsystem haben. Diese aktuelle Situation ist es, die C++ meiner Meinung nach noch zu einem starken Entwicklungspotential verhilft, denn sie stellt an den Entwickler dieser neuen Domäne viel höhere Anforderungen als an den reinen Anwendungsentwickler, der es in den neunziger Jahren gewohnt war, C++ zusammen mit der Unterstützung mächtiger Frameworks einzusetzen.

Während in den Neunzigern die verwendeten Methoden lange Zeit auf der klassischen Objektorientierung stagnierten, brachte der ANSI/ISO-Standard Ansätze mit, die in eine ganz andere Richtung weisen, als es die traditionellen C++-Techniken tun. Techniken, deren methodische Anwendung in der Praxis noch längst nicht vollständig erprobt und ausgeschöpft ist und die sich mit Portierbarkeit, Wartbarkeit, Wiederverwendbarkeit und Reduktion von Komplexität befassen. Diese neuen Methoden und die dazugehörige Syntax so darzustellen, dass sie einerseits in einem theoretischen Kontext stehen und andererseits praktisch anwendbar werden, das war mein Ziel für dieses Buch.

Danksagung

Zur Fertigstellung dieses Buches haben einige Freunde beigetragen, denen ich an dieser Stelle ganz ausdrücklich meinen Dank aussprechen möchte. Da ist Marcel Rüdinger, der mit dem Angebot, die Schreibarbeit am Sitz seiner Firma Datronic Portugal in Madeira zu erledigen, das ganze Projekt ans Laufen brachte. Sein Inselleben brachte mich dazu, in See zu stechen. Er ist ein Mensch, der sich Ziele setzen kann, und möglicherweise färbte das ein wenig auf mich ab. In der langwierigen Arbeit mit dem Text und den Beispielen war mir Ekaterina Myrsova eine gleichmütige Begleiterin. Sie gab mir die Kraft zum Durchhalten. Sascha Morgenstern übernahm mit viel Geduld die inhaltliche Korrektur des Textes und der Beispiele und stand für Diskussionen zur Verfügung. Ebenso möchte ich Tobias Geiger und Robert Fichtner danken, die mit ihren kritischen Anmerkungen gegen Ende des Buchprojekts wesentliche Vorschläge zur Verbesserung der Textstruktur lieferten. Nicht zuletzt stand mir bei der Perfektionierung des LaTeX-Satzes Annett Eichstädt bei.

Inhaltsverzeichnis

Vorwort . . . . . **V**

Danksagung . . . . . **VII**

1 Einführung **1**

2 Die Sprache C++ **5**
2.1 Geschichte und Paradigmenwandel 5
2.2 Grundlagen 9
2.2.1 Bezeichner in C++ 12
2.2.2 Der Präprozessor 13
2.2.3 Variablen 21
2.2.4 Standarddatentypen 22
2.2.5 Literalkonstanten 25
2.2.6 Konstanten 26
2.2.7 Aufzählungen 27
2.2.8 Arrays 29
2.2.9 Zeiger 29
2.2.10 Referenzen 36
2.2.11 Typenkonvertierung 37
2.2.12 Ausdrücke 39
2.2.13 Operatoren 40
2.2.14 Anweisungen 44
2.2.15 Kontrollstrukturen 45
2.2.16 Funktionen 49
2.2.17 Funktionsüberladung 54
2.2.18 Funktions-Inlining 56
2.2.19 Makros 58
2.2.20 Dynamische Speicherallokation 59
2.2.21 Strukturen und Klassen 67
2.2.22 Typenkonvertierung mit Konstruktoren 91
2.2.23 Globale, automatische und dynamische Instanziierung 94
2.2.24 Speicherklassen 100
2.2.25 Der Scope-Operator 106
2.2.26 Verschachtelte Typen 110
2.2.27 Die `friend`-Deklaration 111
2.2.28 Statische Methoden und Attribute 112

2.2.29 Vererbung 114
2.2.30 Virtuelle Methoden und Polymorphismus 121
2.2.31 Operatoren der Typenkonvertierung 127
2.2.32 Mehrfachvererbung 134
2.2.33 Virtuelle Vererbung 138
2.2.34 Das Schlüsselwort `const` 141
2.2.35 Operatorüberladung 145
2.2.36 Exception Handling 155

3 Die Objektorientierte Programmierung mit C++ 173
3.1 Der Klassenbegriff 174
3.2 Die Rolle von Patterns und Idiomen 176
3.2.1 Der Iterator 178
3.2.2 Das Zustandsmuster 179
3.2.3 Das Singleton-Muster 187
3.3 Datenstrukturen und Containerklassen 197
3.3.1 Die Liste 198
3.3.2 Der Vektor 208
3.3.3 Dynamische Container und das Problem der Speicherfragmentierung 210
3.3.4 Verfeinerung des Zugriffs durch überladene Operatoren 211
3.4 Arbeiten mit Invarianten 213

4 Generische und generative Programmierung mit C++ 221
4.1 Templates 222
4.1.1 Funktionstemplates 223
4.1.2 Klassentemplates 226
4.1.3 Methodentemplates 227
4.1.4 Instanziierung von Templates 229
4.1.5 Member-Templates 230
4.1.6 Spezialisierung von Templates 231
4.1.7 Partielle Spezialisierung 232
4.1.8 Vorgabeargumente für Templateparameter 235
4.1.9 Abhängige und qualifizierte Namen 236
4.1.10 Explizite Qualifizierung von Templates 241
4.1.11 Barton-Nackman-Trick 242
4.1.12 Das Schlüsselwort `typename` 244
4.1.13 Template-Templateparameter 247
4.1.14 Container mit Templates 251
4.1.15 Smart Pointer mit Templates 255
4.1.16 Projektorganisation mit Templates 261

4.2 Konzepte für den Einsatz von Templates 268
4.2.1 „Policy Based Design“ 270
4.2.2 Idiome für die Template-Programmierung 271
4.2.3 Compiletime-Assertions 272
4.3 Aspektorientierte Programmierung 273

5 Die C++-Standardbibliothek 277
5.1 Die Geschichte der Standardbibliothek 277
5.2 Die Teilbereiche der Standardbibliothek 278
5.2.1 Die Streams 280
5.2.2 Formatierungen auf einem `ostream` 285
5.2.3 Die Manipulatoren 286
5.2.4 Die File-Streams 289
5.2.5 Die String-Streams 290
5.2.6 Die STL 292
5.2.7 Die Stringklasse `std::string` 326
5.2.8 Pseudocontainer für optimiertes Rechnen 332
5.2.9 Autopointer 342

6 Das Softwareprojekt mit C++ 347
6.1 Modularisierung eines C++-Projekts 347
6.1.1 Der Umgang mit Headerdateien 348
6.1.2 Namensräume 353
6.1.3 Das argumentenabhängige Lookup-Verfahren 361
6.1.4 Einige Anmerkungen zum Linker 363
6.1.5 Überladen der Operatoren `new` und `delete` 367
6.2 Persistenz und Serialisierung 374
6.3 Systemprogrammierung mit C++ 377
6.3.1 Einfaches Objektpooling 377
6.3.2 Nebenläufige Programmierung 382

Literatur 395

Abbildungsverzeichnis 397

Listings 399

Index 409

1 Einführung

Dieses Buch verfolgt zwei Zielrichtungen. Es wendet sich einerseits an Entwickler, die einen fundierten Einstieg in die C++-Programmiersprache suchen und sie in Projekten praktisch einsetzen wollen. Andererseits ist dieses Buch eine Art thematisches Nachschlagewerk, das es dem Kenner der Sprache ermöglicht, bestimmte Bereiche nachzuarbeiten und zu vertiefen. Insbesondere derjenige, der sich mit den dahinter liegenden Leitideen befassen möchte, sollte bei diesem Buch auf seine Kosten kommen.

Vor allem der erste Teil des Buchs ist in der Form eines Lehrbuchs geschrieben. Die einzelnen Unterkapitel der Sprachbeschreibung bauen inhaltlich aufeinander auf und führen den interessierten Leser schrittweise an die Sprache C++ heran. Der zweite Teil des Buchs greift verschiedene moderne Techniken der C++-Entwicklung auf und erläutert diese in einer Form, die nicht auf den Zusammenhang mit den Nachbarkapiteln angewiesen ist. Es wird weniger auf die Zusammenhänge der Techniken eingegangen als vielmehr auf die Alleinstellungsmerkmale von diesen. Das Buch ist mit zahlreichen Listings ausgestattet, die das im Text Gesagte veranschaulichen und einen Anknüpfungspunkt für eigene Experimente abgeben. Deshalb finden Sie die meisten größeren Listings als Quellcode zum Download auf der Internetseite:

```
http://www.oop-trainer.de/publications/de/moderncpp
```

Die Sprache C++ hat nun schon eine beachtliche Entwicklungszeit hinter sich. In dieser Zeit übernahm C++ vieles der sich parallel entwickelnden Programmierparadigmen, wie der Objektorientierung und später der generischen und generativen Programmierung. Nicht alles, was auf diesem zeitlichen Weg entstand, war tragfähig. Manches verschwand nach wenigen Jahren schon völlig aus dem Methodenschatz des Entwicklers, wenn sich herausstellte, dass eine Methode nicht mit anderen integrierbar oder auch nur fehlerhaft war[1]. C++ hatte Anteil an diesen Entwicklungen und nahm vieles auf, so dass heute die Sprache Altes und Neues wie keine andere Sprache vereint. Nicht alles davon ist nach dem heutigen Kenntnisstand sinnvoll. Anderes ist wiederum sehr zukunftsweisend, wird aber noch wenig eingesetzt. In den nachfolgenden Kapiteln soll Altes und Neues angesprochen werden. Dabei wird sich der Text am ANSI/ISO-Standard von 1998 orientieren, obwohl der Autor weiß, dass alle breit eingesetzten Compiler den Standard bis dato nicht vollständig unterstützen. Manche von ihnen weichen sogar grob von ihm ab und unterstützen alte AT&T-Standards oder Mischformen zwischen AT&T C++ und ANSI C++. Da das Buch die praktische Anwendung von C++ beschreiben

[1] Wie zum Beispiel die private oder die geschützte Sichtbarkeit in der Vererbung.

soll, kann diese Tatsache nicht ignoriert werden. Es wird also an den wesentlichen Stellen auch Hinweise auf die unterschiedlichen Compiler geben, und es werden Methoden beschrieben, wie kritische Stellen bezüglich einer Unverträglichkeit unterschiedlicher Standards im Code zu vermeiden sind. Es werden also auch die gängigen Produktionscompiler GNU C++, MS VC++, Borland C++ und Metrowerks C++ an entsprechender Stelle angesprochen.

C++ ist eine objektorientierte Programmiersprache. Trotz der Kritik an ihr bezüglich der Umsetzung des OO-Konzepts ist sie vielleicht auch die OO-Sprache, die diesem Paradigma einen breiten Durchbruch in der Akzeptanz ermöglicht hat – schließlich war es C++, das in den 90er Jahren prozedurale Compilersprachen wie Pascal und C in der PC-Programmierung ablöste und damit selbst eine enorme Verbreitung erfuhr. Es ist deshalb für ein solches Buch wie das vorliegende notwendig, auf die objektorientierten Techniken einzugehen. Es kann jedoch keine durchgehende Methodik des objektorientierten Entwurfs geben, zumal der syntaktische Umfang der Sprache C++ den textuellen Raum des Buches zur Genüge in Anspruch nimmt. Die syntaktischen und technischen Mittel, mit denen C++ die objektorientierte Modellierung unterstützt, werden ausgiebig behandelt. Der Leser sollte also schon Kenntnisse über die Objektorientierte Programmierung mitbringen oder sich parallel erarbeiten, wenn er dieses Buch liest. Neben der Objektorientierung wurde Mitte der 90er Jahre ein weiteres Paradigma in C++ integriert: die Methodik der generischen Programmierung. Seit einigen Jahren wird sie auch der aspektorientierte Ansatz genannt. Dazu war die neue Technik der parametrisierbaren Klassen, der „Templates“, notwendig. Große Bereiche der Standardbibliothek wurden seit der Integration der Templates in C++ auf diese Technik umgeschrieben, da sie große Vorteile gegenüber der objektorientierten aufweist, wenn es darum geht, wiederverwendbaren Code zu schreiben. Dem fortschreitenden Einsatz der generischen Programmierung in C++-Code wird im zweiten Teil des Buchs Rechnung getragen, indem diese Technik in verschiedenen Anwendungsbereichen facettenreich dargestellt wird.

Einen großen Teil im Buch nimmt also die Templateprogrammierung und deren methodische Anwendung ein. Gerade in diesem Bereich hat sich mit der ANSI-Standardisierung 1998 am meisten getan. Dabei werden bis heute, acht Jahre nach dem Standard, die Techniken nur wenig genutzt. Häufig wird nicht erkannt, was in diesen neuen Modellierungstechniken steckt, oder man hat eine gewisse Angst davor, dass die negativen Folgen des Einsatzes die positiven überwiegen könnten. Dazu ist es notwendig die positiven und negativen Konsequenzen des Einsatzes parametrisierbarer Klassen kennenzulernen, um Entscheidungen für eigene Projekte fällen zu können.

Ein durchgängiger Schwerpunkt liegt in dem Buch auf dem Schreiben von portierbarem Code. Das heißt in der Praxis etwas mehr als nur das Einhalten des Standards. Es wurde ja schon angesprochen, dass die gängigen Compi-

ler nur unzureichend standardkonform sind. Deshalb ist es sinnvoll, zwar standardkonform zu programmieren, aber nur solche Standardkonstrukte zu verwenden, die von allen Compilern identisch verstanden werden. Die in diesem Sinne wesentlichen Problemkonstrukte werden Sie in diesem Buch finden.

Abschließend soll noch gesagt werden, dass die Themen des Buchs vollkommen systemunabhängig sind. Die Programmierbeispiele und der Inhalt des Textes sind mit wenigen Ausnahmen nicht spezifisch für die Programmierung des einen oder anderen konkreten Betriebssystems. Es sind unabhängige Techniken, wie sie der ANSI/ISO C++-Standard von 1998 vorsieht. Die Inhalte können also auf beliebige Projekte angewandt werden, sei es in der Entwicklung einer Anwendung für einen Windows Desktop oder in der Embedded-Entwicklung. Das Verständnis der Leitideen des ANSI/ISO C++-Standards ist heute auf fast allen Plattformen, auf denen C++-Entwicklung durchgeführt wird, eine wichtige Grundlage für ein erfolgreiches Erreichen des Projektziels. Dieses Verständnis möchte das Buch geben.

2 Die Sprache C++

C++ ist inzwischen eine relativ alte Programmiersprache, die viele Brüche in ihrer Geschichte erlebt hat. Vieles, was man heute in C++ findet, ist älteren Programmiermethoden und Ideen zu verdanken, die heute nicht mehr unbedingt aktuell sind. Modernere objektorientierte Sprachen, wie zum Beispiel JAVA, sind in ihrer syntaktischen Struktur weit weniger komplex, da sie weniger unterschiedlichen Paradigmen[1] folgen müssen. In den folgenden Abschnitten soll ein Überblick über die verschiedenen Phasen der Entwicklung von C++ gegeben werden. Damit sollen auch die unterschiedlichen Ideen umrissen werden, die bis heute in der Sprache C++ stecken, ihre Grundlage bilden und sie einerseits mächtig und andererseits an manchen Stellen auch unübersichtlich machen. Für den C++-Entwickler stellt sich immer die schwierige Frage der Auswahl geeigneter Methoden für seine aktuelle Arbeit. Dabei muss zwischen konkurrierenden Leitideen ausgewählt werden und es müssen solche verworfen werden, die nicht mehr aktuell sind.

2.1 Geschichte und Paradigmenwandel

C++ entstand in den frühen 80er Jahren aus der Programmiersprache C heraus. Die Sprache wurde in den Bell Laboratories, die später von AT&T übernommen wurden, entwickelt. Federführender Ideengeber und Entwickler war Bjarne Stroustrup, der diese Rolle über lange Jahre beibehielt. Aus C wurde zunächst ein „C mit Klassen“ entwickelt, um die damals neuen Methoden der Objektorientierung den C-Programmierern zugänglich zu machen. Später, 1983, wurde erstmals der Name „C++“ für die neue Sprache verwendet. Er setzt sich zusammen aus dem Namen „C“ und dem Inkrementoperator „++“ aus C, um anzudeuten, dass C++ über C quasi einen Schritt hinausgeht. Die Objektorientierte Programmierung steckte zu damaliger Zeit noch in den Kinderschuhen. Die Methodenausstattung dieser neuen Leitidee war nach heutigem Maßstab noch etwas dürftig und erst in Entwicklung begriffen. Die Sprachkonstrukte von C++ wurden natürlich an den Stand der damaligen Sofwaretechnik angepasst. Mit der fortschreitenden Technik der OO-Programmierung bekam C++ auch neue sprachliche Strukturen. Die Firma AT&T, die die Bell Laboratories übernahm, standardisierte C++ mit einem eigenen Firmenstandard, der von fremden Compilerherstellern übernommen werden konnte. Die meisten Hersteller von C++-Entwicklungssystemen

[1] Paradigma: Leitidee.

machten sich diese AT&T-Standards zunutze und unterstützten sie. Natürlich boten die verschiedenen Hersteller auch spezifische Features ihrer Compiler an, die über den jeweils gültigen Stand der AT&T-Quasi-Standards hinausgingen. Die wesentlichen Standards von AT&T waren durch die Compiler 1.0, 1.1, 2.0, 2.1 und 3.0 vorgegeben. Die jeweilige aktuelle Standardimplementierung wird bis heute *cfront* genannt. Ein cfront-Compiler ist also konform zum aktuellen Standard. Seit Anfang der 90er Jahre wurde die Standardisierung durch das American National Standardisation Institute angestrebt.

Das für C++ zuständige Komitee brauchte allerdings bis 1998, um einen Standard zu verabschieden. In der Folge der ANSI-Standardisierung wurde der Standard auch zur internationalen ISO-Norm erklärt. Was war nun in den verschiedenen Standardisierungsschritten mit C++ passiert?

In den AT&T Standards der 1er Versionen bekam C++ vor allem die Sprachmerkmale, die zur Kapselung von Daten in Klassen notwendig sind. In der objektorientierten Entwicklung der damaligen Zeit bis 1986 standen in besonderer Weise die Gedanken der Datenkapselung und der Funktionsüberladung im Vordergrund. Vererbung wurde eher zur Anhäufung von Daten und Funktionalität verwendet, um eine Art Wiederverwendung gemeinsamer Funktionalität zu erreichen. Die dafür geschaffenen Sprachstrukturen bestehen in C++ bis heute, auch wenn wir die Vererbung nicht mehr in dem damaligen Sinne anwenden, denn sie hat sich als Sackgasse erwiesen. Dass die Vererbung in C++ eine Sichtbarkeit besitzt[2] und diese auch noch privat ist, hat den beschriebenen Hintergrund.

Mit der AT&T-Version 2.0 fanden die virtuellen Funktionen Einzug in die Sprache C++ und mit ihnen das Konzept der Polymorphie. Man kann daher sagen, dass im Grunde genommen erst diese Version der Sprache dem ähnlich wurde, was wir heute als C++ kennen. Mit der Polymorphie als zentralem Konzept der Objektorientierung wurde C++ überhaupt erst zu einer OO-Sprache. Vorher war das, was man C++ nannte, ein erweitertes C mit etwas besserer Typenkontrolle, anderen Kommentarzeichen und der Möglichkeit Daten zu verstecken. Jetzt erst konnten OO-Entwürfe mit C++ verwirklicht werden, die auf die späte Bindung von Methoden aufbauten und damit den Funktionsfokus zugunsten des Objektfokus aufgaben. OO war in C++ angekommen, wenngleich es an der einen oder anderen Stelle noch etwas hakte. So konnten virtuelle Methoden in Basisklassen deklariert werden, um die Polymorphiebasis für abgeleitete Klassen bereitzustellen. Diese virtuellen Methoden mussten formal aber immer implementiert werden, was inhaltlich natürlich wenig sinnvoll war und den einen oder anderen Entwickler noch zu Fehlentscheidungen verführte („Was die Sprache erzwingt, kann ja so falsch nicht sein! Oder?“).

Erst 1992 in der AT&T-Version 3.0 bekam C++ den OO-Rundschliff. Dazu gehörten in allererster Linie die rein virtuellen Funktionen ohne Implementierung. Dieses sehnlichst erwartete und in Newsgroups vorher schon disku-

[2] Siehe Abschnitt 2.2.21 auf Seite 76

tierte Sprachfeature machte nun allen Entwicklern deutlich, dass es Klassen ohne eigene Funktionalität geben kann. Klassen, die eine reine Schnittstellenfunktion haben. Es gab auch noch Versionen bis 3.3, die aber hauptsächlich Elemente einführten, die mit dem OO-Paradigma nichts mehr zu tun hatten. So wurden die Templates und das Exception Handling in den 3er Versionen von C++ aufgenommen. Anfang der 90er Jahre gab es in der Entwicklergemeinde eine Diskussion um die Anwendung der neuen Templatetechnik. So gab es schon Stimmen, die die Templates befürworteten, da sie doch den Compiler in die Lage versetzten, Code zu erzeugen, statt nur zu übersetzen. Andere befürchteten unbeherrschbare Nebeneffekte – z. B. Aufblähung des erzeugten Codes – und eine zunehmende Komplexität der Sprache. Man war schließlich zu einem gesicherten Wissensstand über die Objektorientierung gelangt, da sollte das Instrumentarium nicht durch völlig fremde Konzepte aufgeweicht oder gestört werden. Insbesondere am Design der für C++ längst überfälligen Containerbibliothek entzündete sich die Diskussion. Dynamische Datencontainer werden in C++ benötigt um 1-zu-n-Relationen zwischen Klassen und Objekten zu modellieren. In den C++-Standardbibliotheken der AT&T-Standards waren keine Containerbibliotheken enthalten, weshalb Compilerhersteller eigene herstellerspezifische Bibliotheken auslieferten. Software, die diese herstellerspezifischen Implementierungen nutzte, war natürlich nicht mehr portierbar, was dazu führte, dass häufig für jede spezielle Software extra Container implementiert wurden. In dieser Zeit entstanden sehr viele – sehr viele! – verschiedene, mehr oder weniger gut funktionierende Container (in jedem Softwareprojekt die eigene verkettete Liste). Solche Container können mit reinen OO-Methoden entwickelt werden, oder aber auch mit den generischen Techniken, die durch die Templates in die Sprache eingeführt wurden. Den Streit entschied A. Stephanov[3] mit seiner Standard Template Library zugunsten der generischen Technologie. Diese Bibliothek beruht ganz auf dem Einsatz der C++-Templates und war frei verfügbar. Die STL ist gegenüber entsprechenden OO-Containerbibliotheken von überzeugender Eleganz und Flexibilität. Ein weiterer Vorteil ist die gute Beherrschbarkeit der Laufzeitaspekte beim Einsatz der Bibliothek. Dies alles zusammengenommen führte zunächst dazu, dass die generischen Templatetechniken anerkannt wurden.

Eine weitere Folge war die Aufnahme der STL in den ANSI/ISO-Standard. Die Arbeiten um die Standardisierung veränderten die STL nicht wesentlich. Die STL ist seitdem in leicht angepasster Form Bestandteil der C++-Standardbibliothek. Außerdem verdrängten generische Technologien ihre OO-Pendants aus einigen Bereichen der Bibliothek. Mit gewisser Berechtigung kann man sagen, dass mit der Integration der Templatetechnologie in C++ nun Methoden zur Verfügung stehen, die die lang versprochene Wiederverwendbarkeit Wirklichkeit werden lassen. Die OO-Methoden sind dazu nur marginal in der Lage. Es kam auch das Konzept der Namensräume mit

[3] Der Autor der STL, Alexander Stephanov, wurde durch Hewlett Packard und SGI für deren Entwicklung unterstützt. Über Server bei HP stand die STL schon 1994 zum Download bereit.

in den Standard. Außerdem wurden neue Typenkonvertierungen definiert. Die Templatetechniken wurden mit dem ANSI/ISO-Standard weiter verfeinert und die syntaktischen Möglichkeiten nahmen zu. Sie haben mit dem Standard eine Komplexität erreicht, die das bloße Erlernen der Syntaxregeln enorm erschweren. Dazu kommt noch, dass mit der Verabschiedung des Standards praktisch kein Hersteller in der Lage war, einen Compiler mit korrekter Standardkonformität zu liefern. Lange Zeit stand praktisch nur ein Experimentalcompiler, der Comeau-C++-Compiler, zur Verfügung. Der erste nahezu standardkonforme Produktivcompiler war der Metrowerks Code Warrior, der vor allem für die Macintosh-Plattform verwendet wurde. Eine ähnlich gute Unterstützung des Standards erreichte der GCC-C++-Compiler mit seinen 3er Versionen. Erst mit den stabilisierten Versionen dieses Compilers 2005 stand eine standardkonforme Entwicklungsplattform in der Breite zur Verfügung. Es kommt noch eine weitere Schwierigkeit für die Anwendung des Standards hinzu: in den ANSI/ISO-Standard wurde Exception Handling als integraler Bestandteil aufgenommen. Dabei wurde der `new`-Operator zur Speicherallokation neu definiert. In den AT&T-Versionen von C++ liefert der `new`-Operator einen Nullzeiger zurück, wenn die Allokation fehlschlägt. In der ANSI-Version wirft `new` eine Exception, um einen Fehlschlag anzuzeigen. Diese Änderung ist derart tiefgreifend, dass eine Abwärtskompatibilität standardkonformer Compiler nicht mehr gegeben ist. Wenn Softwareprojekte alte Codeteile verwenden, kann damit der Standard nicht zur Anwendung kommen, da sonst die alte Fehlerbehandlung deaktiviert wird und es zu undefinierten Laufzeitzuständen kommen kann – fast schon zwangsläufig kommen muss. Alter und neuer Code vertragen sich nicht. Alter Code muss zunächst umgeschrieben werden, um mit neuem standardkonformen Code zu harmonisieren[4]. Viele erfolgreiche Bibliotheken sind nach einem AT&T- oder verwandten Standard geschrieben und werden weiter verwendet[5]. Ein weiteres Problem besteht in den Gewohnheiten der Entwickler. Um standardkonform zu entwickeln müssen sie Exception Handling als integralen Bestandteil ihrer Software begreifen, was einen Bruch mit alten Gewohnheiten und ganz neue Programmstrukturen erfordert. Diese Gründe zusammengenommen verhindern acht Jahre nach der Verabschiedung des Standards immer noch seine breite Akzeptanz.

Heute ist schon die nächste Version des ISO-Standards in Diskussion. Dabei wird auch die Aufnahme von freien Bibliotheken, wie z. B. der boost-Library diskutiert. Zu hoffen bleibt allerdings, dass Verhaltensweisen des Sprachkerns nicht mehr redefiniert werden, wie es bei der letzten Standardisierung geschah, da das einer zügigen Annahme eines Standards erneut im Wege stehen könnte.

[4] Es ist im ANSI/ISO-Standard zwar ein Migrationspfad für die alte Anwendung des new-Operators vorgesehen – die „nothrow"-Variante –, jedoch ist eine Portierung alten AT&T-konformen Codes deutlich aufwändiger als die reine Anpassung der Verwendungen des Operators new.

[5] Ein prominentes Beispiel dafür ist sicher die MFC.

2.2 Grundlagen

C++ ist eine reine Compilersprache. Der Quelltext muss durch einen Compiler in ausführbaren Code übersetzt werden, damit ein Programm ablaufen kann. Der Quelltext wird für den Ablaufvorgang der Software nicht mehr gebraucht. Dieses traditionelle Konzept teilt C++ mit älteren prozeduralen Sprachen wie Fortran, Pascal und natürlich auch C. Von C hat C++ die typische Zusammenarbeit zwischen Compiler und Linker geerbt. Nicht jede Compilersprache braucht einen separaten Linker. Das C-Konzept der zweistufigen Bearbeitung bringt jedoch eine sehr flexible Möglichkeit zur Modularisierung von Softwareprojekten mit sich. In der ersten Stufe übersetzt der Compiler ein Quelltextmodul zu einer Objektdatei. In der zweiten Stufe bindet der Linker verschiedene Objektdateien – und damit die Funktionen der verschiedenen Module – zur lauffähigen Software zusammen. C++ wird ebenso durch einen Compiler zu Objektdateien übersetzt, die später durch einen Linker gebunden werden. Dadurch lässt C++ große Freiheiten bei der Aufteilung eines Softwareprojektes in Module. Technisch ist die Flexibilität der Modularisierung eines Softwareprojektes sehr hoch. Es bleibt dem Entwickler überlassen, wie er mit dieser Flexibilität umgeht. Im Abschnitt 6.1 auf Seite 347 wird auf die Modularisierung näher eingegangen. Zunächst sollen jedoch einfache Beispiele gegeben werden, bei welchen die Modulaufteilung noch keine Rolle spielt.

Ein einfaches erstes Beispiel zum Einsatz des Compilers

Fangen wir also mit einem ganz einfachen, für einen Anfang mit einer Programmiersprache sehr typischen „Hello world“ an:

Listing 2.1. Erstes Beispiel

```
#include <iostream>

int main()
{
  std::cout << "Hello world" << std::endl;
  return 0;
}
```

Wenn man mit einem beliebigen Texteditor diesen Text erfasst und unter einem beliebigen Namen mit der Endung `.cc` oder `.cpp` abspeichert, hat man schon einen echten C++-Quelltext. Schreiben Sie zum Beispiel „`prg1.cpp`“. Der Quelltext stellt ein einfaches Programm dar, das die Ausgabe „Hello world“ auf die Standardausgabe schreiben soll. Das Beispiel kann also unter einer textorientierten Shell zum Laufen gebracht werden. Sie brauchen eine beliebige Textshell unter UNIX, Linux oder MacOS. Wenn Sie mit Win-

dows arbeiten, nehmen Sie einfach die DOS-Eingabeaufforderung. Natürlich muss ein C++-Compiler auf dem System installiert sein, damit das Programm übersetzt werden kann. Wichtig ist auch, dass der Compiler aus der Kommandoumgebung, in der Sie arbeiten, erreichbar ist. Auch bei einem installierten Compilersystem müssen nicht immer alle wichtigen Pfade systemweit gesetzt sein. Manche Hersteller legen dazu eine kleine Batch-Datei ihrem Compiler bei, damit die Systempfade damit gesetzt werden können. Dazu muss die Batch natürlich innerhalb der Kommandoumgebung aufgerufen werden. Für manche Compiler muss man eine solche Batch-Datei auch selbst schreiben. Da es von Hersteller zu Hersteller sehr unterschiedlich sein kann, was vorhanden ist und was nicht, und da es darüber hinaus noch auf jedem Betriebssystem andere Verfahren der Batcherstellung gibt, kann dieses Installationsproblem an dieser Stelle nicht hinreichend behandelt werden. Dazu nur drei Regeln (wobei `compiler_root` für das Verzeichnis steht, in dem Ihr Entwicklungssystem installiert wurde):

1. Der Compiler und der Linker sind ausführbare Tools, die im Suchpfad der Umgebung gefunden werden müssen. Normalerweise `compiler_root/bin`.
2. Der Compiler muss das Verzeichnis mit den Includedateien finden. Normalerweise `compiler_root/include`.
3. Der Linker muss das Verzeichnis mit den Bibliotheksdateien finden. Normalerweise `compiler_root/lib`.

Manchmal gehen Compilerhersteller etwas unterschiedliche Wege, um ihren Tools die typischen Orte ihres Arbeitsmaterials mitzuteilen. Das können Umgebungsvariablen sein oder auch Konfigurationsdateien im Compilerverzeichnis. Wenn Sie noch nie mit einem C- oder C++-Compiler zu tun hatten, sollten Sie die erste Installation am besten von jemandem durchführen lassen, der es mindestens schon einmal gemacht hat. Gehen wir also von einem installierten und in Ihrer Kommandoumgebung bekannten Entwicklungssystem aus. Im Allgemeinen ruft man den Compiler auf und übergibt ihm den Quelltextnamen in der Kommandozeile. Also:

```
g++ prg1.cpp
```

für den GNU-Compiler unter Linux oder UNIX.

```
bcc32 prg1.cpp
```

für den Borland-C++-Compiler unter Windows.

```
cl /GX prg1.cpp
```

für den MS-Visual C++-Compiler unter Windows.

Auf manchen Systemen wie zum Beispiel QNX heißt der Aufruf des C++-Compilers `CC`. Also: `CC prg1.cpp`.

Dabei sollte nun auf den Unices eine Datei `a.out` entstanden sein. Bei den Windows-Compilern müsste eine Exedatei `prg1.exe` entstanden sein. Diese Dateien sind ausführbar und können direkt von der Kommandozeile aus gestartet werden.

Also:

```
./a.out
```

für Unix und Linux bzw.

```
prg1
```

für Windows.

Wir haben den Aufruf des Compilers kennen gelernt, der eine ausführbare Datei produziert. Ganz ist dieser Satz nicht richtig, denn irgendwie muss ja noch der in den vorangegangenen Abschnitten beschriebene Linker mit von der Partie sein. In unserem ersten Beispiel haben wir den Linker mit dem Compiler aufgerufen. Die genannten Compilerimplementierungen vereinfachen den Einstieg dadurch, dass bei einem direkten Aufruf des Compilers nach dessen Durchlauf noch der Linker gestartet wird, um die Arbeit zu Ende zu bringen. Wenn man die beiden Arbeitsschritte voneinander entkoppeln möchte – oder muss –, kann man das dem Compiler mit einem Kommandozeilenargument mitteilen. Die entsprechenden Outputs des reinen Compileprozesses finden sich meistens auch bei einem integrierten Compiler-Linker-Lauf im aktuellen Verzeichnis. Sie tragen die Endung `.obj` bei den meisten Windows-Compilern und `.o` bei den Compilern der Unices. Auch diese Zwischenprodukte werden für den Ablauf des Programms nicht mehr gebraucht.

Um dieses erste Programmbeispiel nicht ganz unerklärt zu lassen, sollen die einzelnen Zeilen noch kurz besprochen und entsprechende Verweise auf spätere Kapitel eingefügt werden. Mit der Zeile „`#include <iostream>`“ haben wir eine typische Präprozessoranweisung, wie sie im Abschnitt 2.2.2 erklärt ist. Die Includeanweisung fügt die Datei „`iostream`“ in den von uns geschriebenen Programmquelltext ein. Wo er die Datei `iostream` findet? Sie befindet sich im Allgemeinen im Includeverzeichnis des Compilers. Was es mit dieser Datei auf sich hat, wird im Abschnitt 5.2.1 auf Seite 280 beschrieben. An dieser Stelle sei nur gesagt, dass darin die Definitionen für `std::cout` und `std::endl` zu finden sind. Die Zeile „`int main()`“ und der dazugehörige Block, der durch die geschweiften Klammern begrenzt wird, definiert die Hauptfunktion des Programms. Die Funktion heißt `main`, das Wörtchen `int` zeigt an, dass die Hauptfunktion einen ganzzahligen Wert zurück gibt (in diesem Fall an das aufrufende System), und die einfachen Klammern zeigen an, dass es sich bei `main` überhaupt um eine Funktion handelt. Dazu mehr im Abschnitt 2.2.16 auf Seite 49. Jedenfalls springt die Programmausführung vom Betriebssystem aus zuallererst in die Funktion `main()` und führt die darin enthaltenen Anweisungen aus. Im Beispielprogramm sind das nur zwei. Die Ausgabe auf der Konsole „`std::cout << "Hello world" << std::endl;`“ und die Returnanweisung der Funktion „`return 0;`“, die vereinbarungsgemäß etwas – hier die Null – zurückliefert. Vereinbarungsgemäß deshalb, weil die Funktion `int main()` ja

einen ganzzahligen Rückgabewert in ihrer Deklaration verspricht. Momentan sehen wir von dieser Werterückgabe an das Betriebssystem nichts.

Da jetzt die grundsätzliche Funktionsweise des Compiler-Linker-Gespanns geklärt ist, soll nun der Compilerlauf in seinen Einzelheiten genauer betrachtet werden. Das nächste Kapitel beschreibt die sprachlichen Möglichkeiten, den Quelltext selbst mit Hilfe des Compilers zu manipulieren.

2.2.1 Bezeichner in C++

Es gibt in C++ Schlüsselworte und Operatoren, die den Sprachumfang darstellen. Alles andere in einem Programm muss man selbst definieren und benennen. Auch wenn noch nicht eingeführt wurde, was die Dinge für eine Bedeutung haben, so sollen sie hier doch schon einmal aufgezählt werden: Bezeichnen muss man Konstanten, Variablen, Funktionen, Namensräume, Strukturen, Enums, Unions, Klassen und Klassenmethoden. Welche Namen dafür gewählt werden, ist weitgehend dem Programmierer überlassen. Natürlich sollte er solche Namen finden, die im jeweiligen Zusammenhang einen „Sinn" ergeben. Ganz frei ist er aber beim Erfinden der Bezeichner nicht. Es gibt ein paar Regeln, die für alle Bezeichner gelten, die in einem Programm eingeführt werden können:

1. Bei Bezeichnern in C++ werden Groß- und Kleinschreibung unterschieden. Das heißt, sie sind case-sensitiv. Die beiden Bezeichner `Prozentwert` und `prozentwert` bezeichnen also unterschiedliche Dinge.
2. Bezeichner bestehen aus Buchstaben und können auch Ziffern enthalten.
3. Beginnen muss ein Bezeichner immer mit einem Buchstaben des lateinischen ASCII-Zeichensatzes oder mit dem Unterstrich _. Der ASCII-Zeichensatz umfasst das Alphabet ohne nationale Sonderzeichen. Also `a`, `b`, `c`, `d`, `e`, `f`, `g`, `h`, `i`, `j`, `k`, `l`, `m`, `n`, `o`, `p`, `q`, `r`, `s`, `t`, `u`, `v`, `w`, `x`, `y`, `z` und deren groß geschriebene Varianten `A` – `Z`. Die deutschen Sonderzeichen `ä`, `ö`, `ü` und `ß` sind beispielsweise nicht erlaubt. Das gleiche gilt für die dänischen, die isländischen usw.
4. Erst nach dem ersten Zeichen dürfen auch Ziffern verwendet werden. Also `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` und `9`. Der Bezeichner `a42` ist ein gültiger Bezeichner. Dagegen ist `42a` nicht gültig.
5. Alle Zeichen in einem Bezeichner sind signifikant. Sie werden also zur Erkennung und Unterscheidung herangezogen. Es gibt keine Größenbegrenzung[6].

[6] Das war in älteren C++-Dialekten einmal anders. Dort galten 32 oder 256 Stellen im Bezeichner als signifikant. Alles was darüber hinausging wurde einfach abgeschnitten. Auch heute sind noch solche Compiler im breiten Einsatz, die nach diesem alten Verfahren arbeiten. Natürlich gelten auch bei Compilern, die dem ANSI/ISO-Standard entsprechen, technische Grenzen. Ein Bezeichner sollte also nicht den ganzen Hauptspeicher füllen.

6. Schlüsselworte dürfen nicht als Bezeichner verwendet werden.
7. Es gibt noch einige andere durch den Compiler vordefinierte Bezeichner, die nicht verwendet werden dürfen. So sind beispielsweise einige Konstanten oder Makros durch den Compiler vordefiniert.

Bei der Wahl von geeigneten Bezeichnern müssen die vorangegangenen Regeln beachtet werden. Die Regeln sind allerdings so geartet, dass die Einhaltung einigermaßen intuitiv erfolgen kann. Es kommt selten vor, dass man mit ihnen in Konflikt gerät.

2.2.2 Der Präprozessor

Der Präprozessor ist eine Phase des Compilers, die Textersetzungen am Quelltext durchführt. Diese erste Phase hat mit dem eigentlichen Parsen des C++-Codes noch nichts zu tun. Sie ist dem eigentlichen Compilevorgang zeitlich vorgeschaltet und man kann mit ihr den Quelltext für den eigentlichen Compiliervorgang vorbereiten. So kommt es beispielsweise oft vor, dass man Code portabel auf mehreren Plattformen lauffähig hält, indem man Teile, die nur für eine bestimmte Plattform Gültigkeit besitzen, durch Präprozessoranweisungen für eben diese Plattform inkludiert. Andere Teile können durch den Präprozessor herausgeschnitten werden. Es gibt viele Gründe dafür, warum man Textmanipulationen am Quelltext durch den Compiler durchführen lässt.

Nur die Präprozessorphase kann man für solche Textoperationen nutzen. Man programmiert sie mit speziellen Anweisungen, den sogenannten Präprozessordirektiven[7]. Die Präprozessorphase gibt es auch in der Sprache C und sie wurde von dort nach C++ übernommen (die Präprozessorbefehle unterscheiden sich auch zwischen den zwei Sprachen kaum). Man erkennt die Präprozessordirektiven an der Raute # vor dem Schlüsselwort. Es gibt nicht viele verschiedene Präprozessordirektiven. Sie lassen sich in der folgenden Liste zusammenfassen:

```
#define
#undef
#if
#ifdef
#ifndef
#elif
#else
#endif
```

[7] In den meisten Fällen ist der Präprozessor ein eigenes Tool, das durch den Compiler selbst aufgerufen wird. Daher kann man einen solchen Präprozessorlauf auch ohne einen ganzen Compilerlauf anstoßen, um die Ergebnisse bestimmter Präprozessordirektiven zu überprüfen.

```
#include
#line
#error
#pragma
```

Dazu kommen noch die beiden Operatoren # und ##, die für die Bildung von symbolischen Namen gebraucht werden.

Die Präprozessordirektive #define

Eine sehr einfache Anwendung des Präprozessors findet sich in der Definition von Konstanten durch die #define-Anweisung.

```
#define MAXIMUM    1000000
```

Diese Zeile stellt für den Präprozessor die Anweisung dar, alle Zeichenketten im nachfolgenden Text, die „MAXIMUM“ lauten, durch die Zeichenkette „1000000“ zu ersetzen. Damit wird eine symbolische Konstante geschaffen, die an beliebig vielen Stellen des Quelltextes verwendet werden kann. Vor dem eigentlichen Parsevorgang des Compilers ersetzt der Präprozessor die symbolische Konstante MAXIMUM durch die numerische 1000000. Die #define-Anweisung kann noch wesentlich mehr, als nur Konstanten zu definieren. Mit ihr lassen sich Makros definieren, die im Abschnitt 2.2.19 auf Seite 58 erklärt werden.

Die Präprozessordirektive #include ..

Ebenso leicht ist die #include-Anweisung zu verstehen. In fast allen C- und C++-Programmen findet man am Anfang der Quellcodedateien diese Art der Präprozessordirektiven:

```
#include <iostream>
```

Mit dieser Anweisung fügt der Präprozessor die Textdatei, die in den spitzen Klammern genannt ist, in den aktuellen Code ein. Im Folgenden wird diese Textdatei „Includedatei“ oder „Headerdatei“ genannt. Es wird dabei einfach der Text der Quellcodedatei um den Text der Includedatei[8] an der aktuellen Stelle erweitert. Es ist wirklich nichts Weiteres. Dem Compiler ist es an der Stelle der Includeanweisung vollkommen egal, was in der einzufügenden Includedatei steht. Die Wirkungsweise ist einfach auf die Einfügung des Textes beschränkt, der in der aufgeführten Datei enthalten ist. Spätere Compilephasen nach dem Präprozessorlauf überprüfen das Eingefügte dann syntaktisch. Die spitzen Klammern, die den Dateinamen umschließen, haben die Wirkung, dass der Compiler in einem vorher eingestellten Pfad nach der

[8] Das ist im Prinzip eine normale Quellcodedatei. Allerdings enthält sie für gewöhnlich nur bestimmte Definitionen. In folgenden Abschnitten wird näher darauf eingegangen werden.

angegebenen Datei sucht. In diesem Pfad stehen für gewöhnlich die Bibliotheksheaderdateien. Beim Einsatz mehrerer Bibliotheken kann man den Pfad erweitern. Das geschieht in Abhängigkeit vom eingesetzten Entwicklungssystem in Konfigurationsdateien oder in Umgebungsvariablen. Dem Compiler selbst kann man auch einen Parameter übergeben, der den genannten Pfad setzt. Dieser Kommandozeilenparameter beginnt für gewöhnlich mit `-I`. Möchte man eine Headerdatei vom gleichen Verzeichnis einfügen, aus dem auch die Quellcodedatei stammt, in der die Includeanweisung steht, so verwendet man Anführungszeichen.

```
#include "Definitionen.h"
```

Von diesem Wurzelverzeichnis des aktuellen Compiliervorgangs kann man auch relative Pfadangaben machen.

```
#include "../../mylibs/include/Definitionen.h"
```

Die Headerdateien haben für gewöhnlich die Endung `.h`, `.hpp` oder `.hxx`. Die Endung `.h` war schon in der Sprache C üblich und wurde für C++ beibehalten. Manchmal möchte man die Dateien als C++-Quelltexte kenntlich machen und verwendet dabei eine der beiden anderen Varianten. Notwendig ist das nicht, dem Compiler ist es egal.

In ANSI-C haben auch alle Bibliotheksheaderdateien die Endung `.h`. In den alten AT&T-Standards für C++ wurden die Dateiendungen beibehalten. Erst mit der ANSI/ISO-Standardisierung wurden die Dateiendungen für Bibliotheksheader gestrichen. Das hat den Grund, dass man C++ für so viele Systeme einsetzbar gestalten wollte wie möglich. Dabei kann man sich nicht immer auf das Vorhandensein von Dateisystemen verlassen, die Dateierweiterungen kennen.

Die Präprozessordirektiven `#if`, `#else`, `#elif` und `#endif`

Die Anweisungen `#if`, `#else`, `#elif` und `#endif` werden zur so genannten bedingten Compilierung verwendet. Mit ihnen findet ein „Zurechtschneiden" des Quelltextes im Präprozessorlauf statt. Bevor der Quelltext der eigentlichen syntaktischen Überprüfung unterzogen wird, werden Teile aktiviert bzw. deaktiviert, indem sie eingefügt oder ausgeschnitten werden. Dazu werden mit Hilfe dieser Präprozessordirektiven Konstanten abgefragt, die zur Compilezeit schon feststehen.

Listing 2.2. Verzweigungen im Präprozessorlauf

```
#if   DEBUG == 1
  void TraceOut( char *txt )
  {
    std::cerr << txt << std::endl;
  }
```

```
#elif DEBUG == 2
  void TraceOut( char *txt )
  {
    whiteToFile( "TRACES.TXT"', txt );
  }
#else
  inline void TraceOut( char * )
  {
  }
#endif
```

In dem vorangegangenen Beispiel werden drei Quellcodeabschnitte zur Auswahl gegeben. Je nach dem Wert der Konstanten `DEBUG` wird einer der drei Abschnitte gewählt und übersetzt. Die anderen Abschnitte sondert der Präprozessor schon vor dem eigentlichen Übersetzungsvorgang aus.

Die Anwendung der Präprozessoranweisung `#if` zieht nicht zwingend `#else` oder `#elif` nach sich. Beide sind optional.

Die Präprozessordirektiven `#ifdef` und `#ifndef`

Die beiden Anweisungen `#ifdef` und `#ifndef` sind erweiterte Versionen der Anweisung `#if`. Sie fragen ab, ob ein Bezeichner durch den Präprozessor definiert wurde oder nicht. Man kann sie auch mit der Grundform umschreiben.

Listing 2.3. Verzweigungen im Präprozessorlauf anhand einer Definition

```
#ifdef BIGENDIAN
 #define HIGHBYTE RIGHTBYTE
 #define LOWBYTE  LEFTBYTE
#else // LITTLEENDIAN
 #define HIGHBYTE LEFTBYTE
 #define LOWBYTE  RIGHTBYTE
#endif
```

In dem Beispiel in Listing 2.3 werden zwei Definitionen in Abhängigkeit einer schon existierenden Definition unterschiedlich definiert. Solch ein Beispiel ist durchaus realistisch. Es bleibt aber noch ganz im Bereich der Präprozessordirektiven. Mit der Anweisung `#if` kann man das Beispiel folgendermaßen umformulieren:

Listing 2.4. Die Anwendung des Präprozessoroperators `defined`

```
#if defined(BIGENDIAN)
// ...
```

Das nächste Beispiel präpariert ein Stückchen Code unterschiedlich, je nachdem, ob es von einem C- oder einem C++-Compiler verarbeitet wird.

Listing 2.5. Beispiel: bedingte Compilierung

```
#ifdef __cplusplus
extern "C" {
#endif

int f()
{
  // ...
}

#ifdef __cplusplus
}
#endif
```

Auch dieses ist ein sehr realistisches Beispiel. Bei C++-Compilern ist die symbolische Konstante `__cplusplus` vordefiniert (siehe Abschnitt 2.2.2 auf Seite 20). Das führt in dem Beispiel dazu, dass für einen C++-Compilevorgang zwei kleine Codeschnipselchen mehr compiliert werden als für C[9].

Etwas anders ausgedrückt, kann man den Effekt auch so erreichen:

Listing 2.6. Beispiel: bedingte Compilierung

```
#ifndef EXTC
 #ifdef __cplusplus
  #define EXTC extern "C"
 #else
  #define EXTC
 #endif
#endif // EXTC

EXTC int f()
{
  // ...
}
```

Die Anweisung `#ifndef` negiert die Bedeutung von `#ifdef` einfach. Das `n` steht für „not“. Das Beispiel in Listing 2.6 kann auch mit `#if` reformuliert werden:

Listing 2.7. `#ifndef` anders ausgedrückt

```
#if !defined(EXTC)
 #if defined(__cplusplus)
  #define EXTC extern "C"
```

[9] Die hier gezeigten bedingten Codefragmente `extern „C“` mit den dazugehörigen geschweiften Klammern werden für die Integration von C- und C++-Code gebraucht.

```
 #else
  #define EXTC
 #endif
#endif // EXTC
...
```

Ein weiteres Beispiel demonstriert, wie man zur Compilezeit feststellen kann, ob für ein 16-Bit- oder für ein 32-Bit-System compiliert wird, und legt dann die maximale Größe eines Puffers fest:

Listing 2.8. Detektion zur Compilezeit

```
#if sizeof(void*)==2 // 16 Bit
 #define MAXBUFFERSIZE 32000
#else
 #define MAXBUFFERSIZE 10000000L
#endif
```

Dazu wird der Operator `sizeof` verwendet. Dieser Operator wird im Abschnitt 2.2.4 auf Seite 24 erklärt und liefert immer ein konstantes Ergebnis.

Die Direktive `#undef` macht eine Definition rückgängig. Man hebt mit der Anweisung `#undef` Dinge auf, die mit `#define` definiert wurden. Das sind symbolische Konstanten und Makros. Anzuwenden ist `#undef` wie eine `#define`-Anweisung.

```
   #undef Bezeichner
```

Die Präprozessordirektive `#error`

Die Direktive `#error` führt zu einem sofortigen Abbruch des Compilevorgangs und gibt eine Fehlermeldung aus.

Listing 2.9. Abbruch des Compilerlaufs bei Fehlerfall

```
#if   sizeof(void*)==2 // 16 Bit
 #define MAXBUFFERSIZE 32000
#elif sizeof(void*)==4 // 32 Bit
 #define MAXBUFFERSIZE 10000000L
#else
 #error Kein 16 oder 32-Bit System!
#endif
```

Diese `#error`-Direktive wird dazu verwendet, den Compilevorgang abzubrechen, wenn nicht die benötigten Bedingungen für eine erfolgreiche Übersetzung detektiert werden. Der Text in der Argumentenzeile der Anweisung wird dabei durch den Compiler auf der Standardausgabe ausgegeben.

Die Präprozessoranweisung #pragma

Die Anweisung #pragma ist zwar selbst eine Standarddirektive, ihre Funktion ist es jedoch, nichtstandardisierte Einstellungen spezifischer Compiler vorzunehmen. Die Argumente der Direktive sind also jeweils herstellerspezifisch. Wenn ein Compiler die Argumente einer #pragma-Direktive nicht erkennt, muss er sie ignorieren. Das ist eine Standardvorgabe. Welche Einstellungen durch Pragmas beeinflusst werden können, ist absolut abhängig von den technischen Notwendigkeiten eines Zielcompilers oder vom Erfindungsreichtum des Compilerherstellers. So erzwingt die folgende Pragmaanweisung beim MS-Visual C++ 6.0-Compiler, dass die Warnung mit der Nummer 164 als Fehler behandelt wird.

```
#pragma warning( error : 164 )
```

Für den gleichen Compiler hat die folgende Pragmaanweisung die Bedeutung, dass das Funktioninlining auf eine bestimmte Aufruftiefe begrenzt wird. Ruft also eine Inline-Funktion eine andere Inline-Funktion, und so weiter, werden diese Funktionen nur bis zur der Aufrufebene inline expandiert, die in der Pragmaanweisung genannt wurde.

```
#pragma inline_depth( [0... 255] )
```

Diese Beispiele sollen zeigen, wie compilerabhängig Pragmaanweisungen sind. Will man sie für einen bestimmten Compiler verstehen, muss man seine Dokumentation zu Rate ziehen. Jeder Compiler hat einen großen Satz von möglichen Einstellungen und oft auch bestimmte Funktionen, die nur bei diesem anzutreffen sind.

So erzeugt beispielsweise die Quelltextzeile

```
#pragma keeka
```

beim Borland C++-Compiler Version 5.5.1 für Windows die folgende Ausgabe auf der Konsole:

```
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
keeka.cpp:
 /\__/\
| -  - |
* ^--^ *
 x=--=x
  |    |
|       |
| |  | |     .
| |__| |\___//
 vv  vv *---*

Keeka: Simply the best darn cat in the Universe.
```

Andere Compiler ignorieren dieses Pragma völlig.

Vordefinierte Makros und symbolische Konstanten

Bestimmte Makros und Konstanten sind durch den Compiler bereits vordefiniert. Sie können bei der Definition eigener Makros behilflich sein, ermöglichen die programmtechnische Aufnahme bestimmter Daten, die mit dem Compilevorgang in Verbindung stehen und unterstützen auch die Implementierung von Fehleranalyse- und Abbruchcodes.

Die standardmäßig vordefinierten Makros und Konstanten sind:

`__LINE__`	Zeilennummer als Dezimalkonstante.
`__FILE__`	Dateiname als Stringliteral.
`__DATE__`	Datum als Stringliteral im Format `mm dd yyyy`.
`__TIME__`	Zeit als Stringliteral im Format `hh:mm:ss`.
`__cplusplus`	In ANSI/ISO C++ von 1998 mit 199711L definiert. Wird in zukünftigen Standards wahrscheinlich entsprechend höhere Werte annehmen.

Neben den standardisierten Makros und Konstanten gibt es eine Unmenge von Vordefinitionen, die herstellerspezifisch sind. Jeder Compiler bringt einen Satz von Makros und Konstanten mit, der auf sein bestimmtes Einsatzgebiet und auf seine spezielle Technik abgestimmt ist. Diese Vordefinitionen sind nur in sehr seltenen Fällen kompatibel.

Die Präprozessoroperatoren # und

Die beiden Operatoren werden gebraucht um symbolische Konstanten oder Makronamen durch den Präprozessor generieren zu lassen. Dabei wandelt # eine Zahl in einen Text und ## fügt zwei Texte aneinander.

Mit dem Makro `#define STR(n) #n` lassen sich durch den Präprozessor Literalkonstanten, die Zahlen repräsentieren, in einen Text konvertieren. Der Ausdruck `STR(42)` wird durch den Compiler in den Text „42“ übersetzt.

Listing 2.10. Die Anwendung des Präprozessoroperators ##

```
#include <iostream>

#define SOURCE "Datei: "##__FILE__
#define TRACE( msg ) std::cerr << \
(SOURCE##" => "##msg)<<std::endl;

int main()
{
  TRACE( "Eine Debugmeldung" );

  return 0;
}
```

2.2.3 Variablen

Alle Daten, die in einem C++-Programm verwaltet werden und damit im Speicher gehalten werden, müssen in C++ typisiert sein. Das heißt, dass man sich vorher über die Art der Daten im Klaren sein muss, bevor man mit ihnen umgeht. Für die Daten werden dann Platzhalter definiert. Im einfachsten Fall – die komplizierteren folgen in den Absätzen weiter hinten in diesem Buch – sind es Variablen, die Daten aufnehmen können.

Sehen wir uns also hier die Variablendeklaration in C++ an. Es können in C++ nur Variablen verwendet werden, die vorher deklariert wurden. Variablen als Platzhalter für irgendwelche Daten, die erst zur Laufzeit feststehen, müssen dem Typ ihrer Daten entsprechen. Es gibt keine Variablen, die wie in BASIC verschiedene Datentypen, also zum Beispiel ganzzahlige Werte oder Texte, aufnehmen können. Die Datentypen, die solchen Variablen zugrunde liegen, nennt man variante Typen[10]. Eine Variable in C++ wird also zuerst mit deklariert – mit dem nötigen Datentyp – und dann benutzt:

```
int i;
```

Die Variable `i` hat den Datentyp `int`, der ganzzahlige Werte in einem bestimmten Wertebereich erlaubt. Der Bezeichner `i` erlaubt nun mit der Variablen umzugehen. Man kann dieser Variablen zum Beispiel einen Wert zuweisen:

```
i = 7;
```

Der Begriff „Variable“ steht wie in der Algebra dafür, dass der Platzhalter mit einer bestimmten Bezeichnung für verschiedene Werte steht. Allerdings gibt es einen großen Unterschied zur Algebra. In einem C++-Programm kann eine Variable zu einem Zeitpunkt immer nur genau einen Wert aufnehmen. In der Algebra kann eine Variable durchaus für mehrere mögliche Werte stehen. Die Mathematik steht außerhalb von Zeit und Raum. In einem Programm sieht es anders aus. Ein Programm läuft in der Zeit ab und ändert dabei seinen internen Zustand. Es werden also die Variablen während der Ablaufzeit des Programms geändert. Zu einem bestimmten Zeitpunkt hat eine Variable also immer genau einen bestimmten Wert.

Technisch gesehen ist eine Variable ein Speicherplatz, der Werte eines definierten Typs aufnehmen kann. Der Typ bestimmt die Größe des Speicherplatzes und auch die Weise, wie der Speicherplatz interpretiert wird. Letzteres muss man so verstehen, dass der Speicher, der ja nur eine Aneinanderreihung von Bytes darstellt, alle beliebigen Werte aufnehmen kann, die man in Bytes speichert. In jede Zelle (Byte) passen ganzzahlige Werte von 0 bis 255. Wenn

[10] So etwas kann zwar auch mit C++ erreicht werden. Dazu muss allerdings erst das Klassenkonzept verstanden werden. Deshalb stelle ich mich zunächst auf den Standpunkt, dass es variante Datentypen in C++ nicht gibt.

beispielsweise Text oder Fließkommazahlen abgespeichert werden sollen, so muss es ein Verfahren geben, die zu speichernde Information in eben diesen Werten abzulegen, die man in Bytes speichern kann. Hinter einem Fließkommatyp, wie zum Beispiel `double`, steht also die Speichergröße, die ein solcher Zahlentyp im Speicher braucht (acht Bytes), und das Verfahren, wie eine solche Kommazahl in dem Speicher untergebracht wird. Eine Zuweisung, wie sie an der Variablen `i` demonstriert wurde, ordnet die Daten in einer dem Typ inneliegenden Weise und legt sie im Speicher ab.

Im Gegensatz zu C können in C++ fast an jeder Stelle Variablen deklariert werden[11]. Wenn man mehrere Variablen des gleichen Typs braucht, kann man die Variablenbezeichner hinter dem Typ durch Komma trennen.

```
int a, b, c;
```

Diese Deklaration legt drei Variablen des Typs `int` an. Außerdem können Variablen an der Stelle der Deklaration gleich mit einem Wert initialisiert werden.

```
int a = 1, b = 2, c = 3;
```

Die Variablen, die innerhalb von Funktionen deklariert werden, liegen auf dem Stack. Das ist ein Speicherbereich, der unter anderem für diese lokalen Daten zur Verfügung steht. Der Stack und die möglichen Orte der Variablendeklaration sind in Abschnitt 2.2.23 auf Seite 94 beschrieben.

2.2.4 Standarddatentypen

C++ ist eine typisierende Programmiersprache. Das heißt, dass alle Daten, die in einem C++-Programm verwaltet werden, einen Datentyp besitzen. Ein Datentyp beschreibt, um welche Art von Daten es sich handelt. Diese Aussage geht durchaus über das hinaus, was in diesem Abschnitt besprochen wird. Hier werden die Grundbausteine der Datentypen besprochen: die einfachen numerischen Typen, die man auch als *Standarddatentypen* bezeichnet. In späteren Abschnitten und Kapiteln werden noch ganz andere Aspekte des Datentyps in der Objektorientierten Programmierung beschrieben.

Zunächst aber zu den Datentypen, die durch die Sprache C++ fertig mitgeliefert werden: die Standarddatentypen. Es gibt vier vorzeichenbehaftete und vier vorzeichenlose ganzzahlige Typen. Die Grundtypen `int`, `short`, `long` sind vorzeichenbehaftet. Man kann das Vorzeichen auch dadurch zum Ausdruck bringen, dass man ihnen das Schlüsselwort `signed` voranstellt. Der vierte vorzeichenbehaftete ganzzahlige Typ ist `signed char`. Von diesen Typen lassen sich die vorzeichenlosen Varianten bilden, indem man ihnen das

[11] In C müssen sie immer am Anfang eines Blocks vor den Anweisungen stehen.

Schlüsselwort `unsigned` voranstellt. Also `unsigned char`, `unsigned short`, `unsigned int` und `unsigned long`. Der Typ `char` ist je nach Voreinstellung des Compilers vorzeichenlos oder -behaftet.

Während die Typen `char`, `short` und `long` feste Breiten besitzen, richtet sich der Datentyp `int` nach der Vorgabe des Compilers, die meistens der Busbreite des Prozessors entspricht. Dieser Typ hat aber mindestens zwei Byte Breite.

Alle Standarddatentypen im Überblick:

Vorzeichenbehaftete Typen		**Vorzeichenlose Typen**	
	int		
oder	**signed int**		**unsigned int**
oder	**signed**	*oder*	**unsigned**
Bereich	systemabhängig entweder wie long oder wie short	*Bereich*	systemabhängig entweder wie unsigned long oder wie unsigned short
	long		
oder	**signed long**		**unsigned long**
Bereich	$-(2^{31}) .. 2^{31} - 1$	*Bereich*	$0 .. 2^{32} - 1$
	short		
oder	**signed short**		**unsigned short**
Bereich	$-(2^{15}) .. 2^{15} - 1$	*Bereich*	$0 .. 2^{16} - 1$
	char		
	Kann signed oder unsigned sein		
	signed char		**unsigned char**
Bereich	$-(2^{7}) .. 2^{7} - 1$	*Bereich*	$0 .. 2^{8} - 1$
	float		
Genauigkeit 32 Bit	$3,4e10^{-38}..3,4e10^{38}$		
	double		
Genauigkeit 64 Bit	$1,7e10^{-308}..1,7e10^{308}$		
	bool		
Bereich	`true` und `false`		

Der Operator `sizeof`

Mit dem Schlüsselwort `sizeof` kann die Größe eines Datentyps in Byte bestimmt werden. Der `sizeof()`-Ausdruck wird immer zur Compilezeit ausgewertet. Das heißt, dass `sizeof(short)` durch den Compiler immer direkt in `2` übersetzt wird. der Ausdruck `sizeof(int)` ist maschinenabhängig, weil der Typ `int` eben abhängig ist. Trotzdem ist er bei einem Compiledurchlauf immer konstant. Der Operator kann auch auf Variablen angewendet werden, liefert aber auch dann nur die Größe des zugrunde liegenden Typs. Er liefert also immer einen konstanten Wert, der nicht erst zur Laufzeit ermittelt werden muss. Das folgende Programm demonstriert die Wirkung des Operators `sizeof`:

Listing 2.11. Anwendung des `sizeof`-Operators

```
#include <iostream>

int main()
{
  std::cout << sizeof(short) << std::endl;
  std::cout << sizeof(int) << std::endl;
  std::cout << sizeof(long) << std::endl;
  std::cout << sizeof(float) << std::endl;
  std::cout << sizeof(double) << std::endl;

  int x = 42;

  std::cout << "Der Wert " << x
            << " wird in einer Variablen mit der Größe "
            << sizeof(x) << " gehalten." << std::endl;

  return 0;
}
```

Das Beispiel aus Listing 2.11 erzeugt folgende Ausgabe:

```
2
4
4
4
8
Der Wert 42 wird in einer Variablen mit der Größe 4 gehalten.
```

2.2.5 Literalkonstanten

An dieser Stelle soll der Begriff der Literalkonstanten eingeführt werden. Literalkonstanten sind Konstanten, die der Compiler ohne ausgewiesene Deklaration als solche erkennt. Das ist vielleicht ein schwerer Satz, der Inhalt ist dafür umso leichter. So ist zum Beispiel eine Zahlenangabe im Quelltext wie zum Beispiel `42` eine Literalkonstante, denn der Compiler weist ihr bereits den Typ `int` zu, ohne dass dies explizit durch den Programmierer gemacht werden muss. Da sich der Wert einer fest codierten Zahl auch nicht ändern kann, handelt es sich um eine Konstante. Das gilt auch für Fließkommazahlen, deren Nachkommastellen in C++ durch einen Punkt von den ganzzahligen Stellen abgetrennt werden. Beispiel: `3.141`.

Den numerischen Literalkonstanten können noch Typenspezifikationen mitgegeben werden, indem man an die Zahl ein entsprechendes Suffix anhängt. Das `u` oder `U` steht für `unsigned`, also vorzeichenlos. Eine numerische Literalkonstante mit einem solchen Anhang wird also als vorzeichenloser Typ aufgefasst: `65000U`. Weitere Suffixe existieren für `long` mit `l` oder `L` und für `float` mit `f` bzw. `F`.

Numerische Literalkonstanten können auch durch Präfixe modelliert werden. Diese Präfixe wirken allerdings nicht auf den Typ der Literalkonstanten, sondern auf die Art, wie sie interpretiert werden. So gibt das vorangestellte Zeichenpaar `0X` oder `0x` an, dass die Zahl hexadezimal interpretiert werden soll. Dabei gelten in einer hexadezimalen numerischen Literalkonstante auch die Ziffern `A` bis `F` (auch diese Ziffern dürfen groß oder klein geschrieben werden). Die Zahl `0x0FUL` beschreibt zum Beispiel die dezimale 16 mit dem Typ `unsigned long`. Sie könnte also auch `16UL` geschrieben werden. Ein weiterer Präfix ist die vorangestellte `0`. Sie erzwingt die oktale Interpretation einer numerischen Angabe. Die erlaubten Ziffern gehen von `0` bis `7`. Die Zahl `010` ist also die dezimale `8`. Mit der vorangestellten `0` ist deshalb Vorsicht geboten[12].

Sonderformen der numerischen Literalkonstanten sind durch die Schlüsselworte `true` und `false` gegeben. Es sind die beiden möglichen Werte des Typs `bool`. Weiter modifizieren kann man die beiden boolschen Literalkonstanten nicht.

Neben den numerischen Literalkonstanten gibt es aber auch die der Textstrings. Texte, die im Quelltext in doppelte Anführungszeichen gesetzt wurden, werden als Textarrays erkannt und entsprechend behandelt. Typisiert sind sie mit `char *const`, was in dem späteren Abschnitt 2.2.9 erklärt wird. Textkonstanten im Quellcode, wie zum Beispiel `"Dies ist ein Text"`, sind also ebenso Literalkonstanten.

[12] Im praktischen Einsatz von C++ zeigt es sich immer wieder, dass Entwickler zur „Verschönerung“ mancher Zahlen im Quelltext eine Null voranstellen möchten ohne dabei zu bedenken, dass sich damit der numerische Wert der Zahl ändert.

2.2.6 Konstanten

Im traditionellen C werden Konstanten mit der Präprozessordirektive `#define` definiert. Dabei führt der Compiler eine einfache Textersetzung durch, ohne den Typ der Daten zu überprüfen. Typisieren kann man die Konstanten dadurch, dass man ihnen einen Spezifizierer anhängt, der den ersetzenden Text für nachfolgende Compilerphasen als Literalkonstante eines bestimmten Standardtyps ausweist.

```
#define OBERGRENZE 1000UL
#define UNTERGRENZE 500UL
```

Konstanten können in C++ auch mit dem Schlüsselwort `const` gebildet werden. Man setzt `const` vor oder nach einer Typenbezeichnung ein und typisiert damit einen Bezeichner. Der Bezeichner repräsentiert die Konstante und muss daher sofort auf seinen Wert gesetzt werden. Also:

```
const unsigned long obergrenze = 1000;
const unsigned long untergrenze = 500;
```

oder

```
unsigned long const obergrenze = 1000;
unsigned long const untergrenze = 500;
```

Grundsätzlich kann in C++ die „alte" Variante aus C verwendet werden, Konstanten zu definieren. Ein Nachteil erwächst aus der fehlenden Typisierung der Bezeichner, wenn man nicht die Möglichkeiten der Literalkonstanten ausnutzt. Vom Gesichtspunkt der Typisierung also ist die echte Deklaration der `#define`-Variante vorzuziehen. Die Deklaration einer Konstante mit dem Schlüsselwort `const` garantiert auch einen Speicherplatz, sollte mittels einer Referenz oder eines Zeigers auf die Konstante zugegriffen werden müssen. An dieser Stelle kommt aber auch schon der Nachteil der echten Deklaration ins Spiel. Da es sich bei den deklarierten Konstanten um Speicher handeln kann, sind diese im gewissen Sinne „teurer". Um die Speicherkosten im Zaum zu halten, ist man auf die sehr unterschiedlich ausgeprägten Optimierungsmöglichkeiten der verschiedenen Compiler angewiesen, oder auf eine manuelle Aufteilung globaler Konstanten in verschiedene Module analog zur Deklaration externer Variablen. An anderen Stellen, wie zum Beispiel innerhalb von Klassen oder Strukturen, lässt sich der Speicherbedarf der deklarierten Konstanten überhaupt nicht reduzieren[13]. Ganz unproblematisch nimmt sich die Deklaration von Konstanten in einem lokalen Funktionsscope aus. Dort sind die besten Optimierungsvoraussetzungen gegeben, die der Compiler im Allgemeinen dank der zusätzlichen Typeninformation auch nutzen kann.

[13] Siehe Abschnitt 2.2.21 auf Seite 72.

Eine weitere in C++ genutzte Möglichkeit Konstanten zu bilden, ist die Definition von Aufzählungstypen. Innerhalb eines `enum`-Typs werden Konstanten aufgezählt. Dabei gibt es in C++ auch verschiedene Situationen, die den Einsatz des Schlüsselwortes `enum` nur für die Deklaration von Konstanten notwendig machen und den ursprünglichen Sinn des Aufzählungstyps in den Hintergrund treten lassen. Es soll aber zuerst die Idee der Aufzählungstypen erläutert werden.

2.2.7 Aufzählungen

Mit dem Schlüsselwort `enum` lassen sich sogenannte Aufzählungstypen definieren. Aufzählungstypen sollen eine genau umrissene Anzahl von Werten annehmen können.

Listing 2.12. Syntax des `enum`-Typs

```
enum STATUS { AUS, AN };
// ...
STATUS status = AN;
// ...
switch(status)
{
  case AN:
    // ...
    break;
  case AUS:
    // ...
    break;
}
```

Im Beispiel ist der Bezeichner `STATUS` der Datentyp und der Bezeichner `status` steht für eine Variable. Der Variablen kann – im Idealfall – nur der Wert `AN` oder `AUS` zugewiesen werden. Das wird vom Compiler allerdings nur teilweise überprüft. Dazu eine Vorüberlegung:

Der Compiler muss aus dem Aufzählungstyp etwas machen, das im Speicher abgelegt werden kann. Dabei sollte er platzsparend vorgehen, um gegebenenfalls optimieren zu können. Die verschiedenen Zustände wird der Compiler mit ganzzahligen Werten belegen. Also handelt es sich im Grunde genommen um einen Integertyp. Die Zustände werden bei null beginnend in aufsteigender Reihenfolge durchnummeriert. Im vorangegangenen Beispiel wurden also mit dem Typ `STATUS` zwei Konstanten `AUS` und `AN` definiert. `AUS` hat den Wert null und `AN` den Wert eins. Was der Compiler tut, kann man auch explizit ausdrücken:

```
enum STATUS { AUS = 0, AN = 1 };
```

Dabei kann man auch andere Werte definieren. Es müssen nicht die sein, die der Compiler vorbelegt hätte. Außerdem müssen die Werte nicht direkt aufeinander folgen und können auch in unsortierter Reihenfolge angegeben werden.

```
enum BUCHUNGSSTATUS { UNGEBUCHT = 0,
                      GEBUCHT = 1,
                      STORNIERT = -1 };
```

Der zugrunde liegende Integerdatentyp wird durch den Compiler anhand der vergebenen Werte bestimmt. Der minimale und der maximale Wert umfassen den Wertebereich, in dem die Ausprägungen des Aufzählungstyps liegen. Damit wird auch grundsätzlich festgelegt, ob der deklarierte `enum`-Typ auf einen vorzeichenbehafteten oder einen vorzeichenlosen ganzzahligen Typ abgebildet wird. Der Wertebereich des Integers wird anhand der Zweierpotenz bestimmt, die alle Ausprägungen aufnehmen kann. Der Datentyp `enum STATUS {AUS,AN}` wird also auf einen 1-Bit-Integertyp abgebildet, da dieser die zwei Zustände aufnehmen kann. Dabei kann der gebildete Integertyp oft mehr Werte aufnehmen als der `enum`-Typ direkt definiert. Es können größere Abstände zwischen den Aufzählungswerten bestehen oder der größtmögliche Wert des Integers durch keinen Aufzählungswert abgedeckt sein. *Der Compiler arbeitet mit einer ganzzahligen Typbeschreibung, die von den Bytegrenzen abweicht.* Diese Typenbeschreibung legt er auch der Typenüberprüfung zugrunde. Dabei sind für Zuweisungen an einen Aufzählungstyp alle Werte zulässig, die der zugrunde liegende Integertyp fassen kann. Ob diese Werte auch in der Liste der Aufzählungswerte genannt wurden, ist nicht ausschlaggebend. Insofern ist die Überprüfung der Aufzählungstypen unzureichend und nur durch den zugrunde liegenden Integertyp bestimmt.

Bei der Deklaration eines Aufzählungstyps fallen – wie im vorausgehenden Abschnitt beschrieben wurde – auch Konstanten an. Man kann die Deklaration auch zur reinen Definition von Konstanten verwenden. Dafür steht auch die Möglichkeit zur Verfügung, einen `enum`-Typ anonym zu deklarieren. Dazu lässt man einfach den Typenbezeichner nach dem Schlüsselwort `enum` weg. Damit ist nur die Konstante als Teil der Definition verwendbar.

```
enum { OBERGRENZE = 10000 };
```

Insbesondere dann, wenn Konstanten Teil einer Klasse sein sollen, ist die beschriebene Technik den anderen in bestimmten Situationen überlegen. Eine Konstante, die mittels einer `enum`-Deklaration definiert wurde, verbraucht nämlich keinen Speicher. Dazu wird aber in späteren Kapiteln mehr zu lesen sein[14].

[14] Siehe Abschnitt 2.2.21 auf Seite 72.

2.2.8 Arrays

Arrays sind Speicherbereiche, die mehrere Einheiten eines bestimmten Datentyps direkt aneinanderliegend beinhalten. So können mit der folgenden Deklaration zehn Variablen des Typs `int` angelegt werden, die sich direkt benachbart im Speicher befinden:

```
int werte[10];
```

Eine solche Deklaration kann global oder lokal erfolgen. Das heißt, dass Arrays sowohl auf dem globalen Namensraum und damit im Datensegment angelegt werden können, als auch im lokalen Namensraum einer Funktion und damit auf dem Stack[15].

Eingeleitet wird die Arraydeklaration vom Typbezeichner, der den Typ der einzelnen Elemente bestimmt. Danach kommt der Name des Arrays gefolgt von den eckigen Klammern, die einen Wert beinhalten, der die Anzahl der Elemente festlegt.

Zugegriffen wird auf die einzelnen Elemente des Arrays mit den eckigen Klammern und der darin enthaltenen Elementnummer. Dabei beginnt man die Zählung mit der 0. Also ist `werte[0]` das erste Element aus dem oben deklarierten Array und `werte[9]` das zehnte und damit letzte Element. Die Elemente aus einem Array sind, sofern sie nicht als Konstanten deklariert sind, Variablen wie andere auch. Sie können also in einem beliebigen Ausdruck verwendet werden.

2.2.9 Zeiger

Die Zeiger[16] sind ein Element der Programmierung, das die traditionelle Sprache C und damit auch C++ von allen anderen Programmiersprachen abhebt. Wenn man C und C++ programmieren will, muss man die Zeiger verstanden haben. Ohne geht es nicht, denn das ganze Konzept ist auf sie aufgebaut. Aus diesem Grund soll hier sehr genau darauf eingegangen werden, was Zeiger eigentlich sind.

Betrachten wir zunächst eine normale Variablendeklaration der Form:

```
int i;
```

oder

```
double d;
```

[15] Auch wenn der deutsche Begriff „Stapel“ gebraucht werden kann, verwende ich lieber den englischen Begriff „Stack“, da er sich in der Praxis durchgesetzt hat.

[16] In Englisch „Pointer“.

Mit einer solchen Deklaration wird Speicherplatz angelegt in der spezifischen Größe des Typs. Das sind für den Typ `int` auf vielen aktuellen Systemen 4 Bytes[17], für `double` sind es 8 Bytes. Je nachdem wo die Deklaration steht, wird die Variable in einem anderen Speicherbereich angelegt. Steht sie auf dem globalen Namensraum oder ist sie statisch in einer Funktion deklariert, dann befindet sie sich im so genannten Datensegment. Ist sie hingegen in einer Funktion als automatische Variable oder als Parameter deklariert, dann befindet sie sich auf dem Stack. Wenn sie als Element in einer Struktur, Klasse oder einer Union deklariert ist, richtet sich der Speicherort nach dem Ort der jeweils angelegten Objekte der zusammengesetzten Datentypen.

Alle diese Speicherbereiche haben eine Gemeinsamkeit: *Sie haben eine Adresse.* Mit dieser Adresse kann der Speicher eindeutig identifiziert werden. Es gibt innerhalb des ablaufenden Prozesses – im Allgemeinen das Programm – keine zwei Speicherzellen mit der gleichen Adresse. Die Adresse ist einfach eine Nummer, die eine Zelle im fortlaufend nummerierten Speicher beschreibt[18]. Dabei ist es egal, ob der Speicher im Stack, im Heap[19] oder im Datensegment liegt. Grundsätzlich kann man also auf Speicherbereiche mit Hilfe der Adresse verweisen.

Wichtig ist bei einem solchen Verweis aber auch die Information, was man an der entsprechenden Speicherstelle erwartet. Denn wenn auf den Speicher zugegriffen wird, muss ja eine bestimmte Anzahl von Bytes in einer bestimmten Weise interpretiert werden. Wenn also eine Adresse auf einen Speicherbereich weitergegeben wird, die eine `int`-Variable enthält, dann müssen an der empfangenden Stelle vier Bytes aus dem Speicher ausgelesen werden und als vorzeichenbehaftete Ganzzahl verstanden werden. Mit diesen Vorüberlegungen ausgestattet können wir uns die erste Zeigerdeklaration in C ansehen.

```
int *p;
```

Ein Zeiger ist eine Adressvariable. Er kann eine Speicheradresse aufnehmen und verweist damit auf einen Speicherbereich[20]. Der Stern (`*`) macht die Variable zu einer Adressvariablen (zu einem Zeiger), der Typ bezeichnet den Inhalt des Speichers, auf den der Zeiger verweisen kann. Mit dem Zeiger

[17] Da der Datentyp `int` systemabhängig ist, kann er auch – auf 16-Bit-Systemen – zwei Bytes umfassen. In seltenen Fällen – auf manchen 64-Bit-Systemen – ist er 8 Bytes groß.

[18] Diese Sicht stellt zwar eine Vereinfachung dar, kann aber für die Applikationsentwicklung so stehen gelassen werden. Aus der Sicht des Betriebssystems stellt sich die Sache unter Umständen etwas anders dar. Wenn das System eine virtuelle Speicherverwaltung besitzt, wird zwischen physikalischem und virtuellem Speicher unterschieden. In einem Prozess kann durchaus ein Adresswert vergeben werden, der dem eines anderen Prozesses gleicht. Die Prozesse sind in diesem Fall aber voneinander abgeschirmt.

[19] Der freie Speicher, der dynamisch allokiert werden kann.

[20] Das Verweisen auf den Speicherbereich erschöpft sich dabei auf das Enthalten der Adresse. Ungefähr so, wie eine Visitenkarte auf eine reale Adresse einer Person verweist. Es ist zunächst kein Vorgang damit verknüpft.

p kann also nur auf einen Speicherbereich verwiesen werden, der int-Daten enthält. Der folgend deklarierte Zeiger kann nur auf double-Daten verweisen:

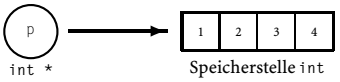
```
double *pd;
```

Ein Zeiger verweist auf einen Speicher, indem er die Adresse des Speichers aufnimmt:

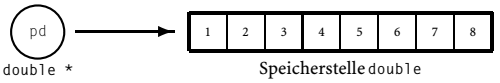
Listing 2.13. Zeigersyntax

```
int i;
int *p;
double d;
double *pd;
/* ... */
p = &i;   // Zuweisung der Adresse von i.
pd = &d;  // Zuweisung der Adresse von d.
```

Das Ampersandzeichen & vor der Variablen i wird in C und C++ „Adressoperator“ oder auch „Referenzierungsoperator“ genannt. Es führt dazu, dass nicht der Wert, sondern die Adresse einer Speicherstelle geliefert wird. Diese Adresse kann dem Zeiger zugewiesen werden. Die Variable p aus Listing 2.13 „zeigt“ damit auf die Speicherstelle, an der i untergebracht ist. Er kann nur auf Speicherstellen zeigen, die einen int-Wert aufnehmen können. Die Typisierung des Zeigers p mit int garantiert, dass nur auf die vier Bytes des int-Wertes zugegriffen werden kann.



Dabei verhält es sich mit Zeigern anderer Typen genauso wie mit den int-Zeigern. Es muss immer der zum Inhalt des Speichers passende Zeiger gewählt werden. Der Zeiger pd in Listing 2.13 zeigt auf eine Speicherstelle, die mit acht Bytes für die Aufnahme eines double-Wertes vorgesehen ist.



Die Zeigervariable ist in ihrer Verweildauer im Speicher absolut unabhängig von dem Speicher, auf den sie verweist. Wird ein Speicherplatz gelöscht, heißt das noch nicht, dass alle Zeiger darauf auch gelöscht werden. Mit diesen Zeigern darf nur nicht mehr zugegriffen werden, da sonst undefinierte Daten gelesen werden oder es zu einem Laufzeitfehler kommt. Im gezeigten Beispiel ist das noch kein Problem, da Speicher und Zeiger im gleichen Gültigkeitsbereich definiert sind.

Ein Zeigerwert kann einem anderen Zeiger zugewiesen werden.

Listing 2.14. Zeigersyntax

```
int i;
int *p1;
int *p2;
/* ... */
p1 = &i;
p2 = p1; /* Die Adresse aus p1 wird p2 zugewiesen. */
```

Wenn man über einen Zeiger auf den Inhalt des Speichers zugreifen möchte, braucht man dazu wieder einen Operator, den Dereferenzierungsoperator (`*`):

Listing 2.15. Zeigersyntax

```
int i;
int *p;
p = &i;
i =7;
printf( "‘%d\n"', *p );
```

In dem Beispiel gibt es eine Variable `i` und einen Zeiger `p`. Der Zeiger wird auf die Variable `i` gerichtet. Die Werte, die `i` annimmt, können nun auch über den Zeiger abgerufen werden – im vorliegenden Fall 7. Innerhalb der `printf()`-Anweisung wird die Variable `i` über den Zeiger `p` mit Hilfe des Dereferenzierungsoperators ausgelesen. Dieser Operator kann auch zum Beschreiben der Variablen verwendet werden:

Listing 2.16. Zeigersyntax

```
int i;
int *p;
p = &i;
*p = 7;
printf( "‘%d\n"', i );
```

Auch in diesem Beispiel gibt die `printf()`-Anweisung den Wert 7 auf der Standardausgabe aus.

Wie weiter oben schon beschrieben, braucht die zugreifende Stelle die Information, in welcher Weise der Speicher interpretiert werden soll. Wenn der Zeiger nicht typisiert wäre, könnte die aufrufende Stelle auch nicht die Größe des zu lesenden bzw. zu schreibenden Speichers bestimmen. Der Typ ist also wichtig und wird vom Compiler überprüft. Der folgende Code ist deshalb nicht gültig:

Listing 2.17. Zeigertypenkonflikt

```
double d;
int *p;

p = &d; /* Compilierfehler! */
```

Im Beispiel passen die Typen der Zeiger nicht zueinander. Während auf der linken Seite ein `int *` erwartet wird, steht auf der rechten Seite ein `double *`. Zwischen diesen Typen kann es keine implizite Konvertierung geben!

Die Verwaltung von Text

Um die Verwaltung von Text in einem C++-Programm zu verstehen, muss man Zeiger und die Arrays verstanden haben. Texte werden in C++ in Form von Arrays verwaltet. Um diesen Verwaltungsaufwand zu minimieren und nicht alles den Programmierer selbst machen zu lassen, gibt es eine Stringklasse in der Standardbibliothek von C++. Diese wird in Abschnitt 5.2.7 ab Seite 326 beschrieben. In diesem Abschnitt sollen Grundlagen gelegt werden, weshalb absichtlich auf den Einsatz der Stringklasse verzichtet wird.

Eine Literalkonstante wie `"Text"` wird durch den Compiler so verarbeitet, dass er ausreichenden Speicher besorgt und dort die Zeichen hineinkopiert. Der dafür nötige Speicherplatz ist allerdings um ein Zeichen größer als der Text in den Anführungszeichen. Nach dem letzten Zeichen wird noch eine binäre Null angehängt, um das Ende eines solchen Strings zu kennzeichnen. Die einzelnen Speicherzellen sind mit `char` typisiert. Mit einem String arbeitet man über Zeiger. Man richtet einen Zeiger beispielsweise auf den Anfang eines solchen `char`-Arrays, um mit dem Text irgendwelche Operationen durchzuführen.

Listing 2.18. C-Strings

```
// ...
char *txt = "Eine Zeile Text";
printf( txt );
// ...
```

Der Code in dem vorangehenden Listing führt zur Ausgabe des Textes auf der Standardausgabe. Die Funktion `printf()` nimmt einen Zeiger auf ein `char`-Array entgegen und gibt Zeichen für Zeichen aus. Dabei achtet sie auf die binäre Null am Ende des Strings, um nicht darüber hinauszulaufen.

Um ganz genau zu sein: Die Literalkonstante der rechten Seite der Zuweisung in Listing 2.18, `"Eine Zeile Text"`, liefert den Typ `char *const`. Sie liefert einen konstanten Zeiger auf ein Array mit Elementen vom Typ `char`. Der variable Zeiger `txt` darf natürlich den konstanten Zeigerwert aufnehmen.

Die Behandlung und Verwaltung von Strings in C++ ist also auf das engste mit den Zeigern und den Arrays verknüpft. Das hat C++ aus C geerbt und exakt so beibehalten, wie es schon in der Vorgängersprache gehandhabt wurde. Erleichterungen erfährt der C++-Programmierer durch die neuen Möglichkeiten der Programmierung mit Klassen. Damit lässt sich auch die Verwaltung von Texten vereinfachen, wie es in Abschnitt 5.2.7 beschrieben wird.

Zeigerarithmetik

Neben der „Schablone“ für den Speicher stellt der Typ des Zeigers auch die Voraussetzung für die Zeigerarithmetik dar. Diese Zeigerarithmetik ist es, die C so sehr von anderen Compilersprachen unterscheidet. C++ hat die Zeigerarithmetik aus C übernommen, weshalb sie hier ausführlich behandelt werden soll.

Betrachten wir zunächst einmal den folgenden Fall:

Listing 2.19. Array-Syntax

```
short buffer[20];
short *p;

p = &buffer[0];
```

Der Zeiger `p` zeigt damit auf das erste Element des Arrays `buffer`. Über `p` kann also auf dieses Element zugegriffen werden. Bildlich sieht das folgendermaßen aus:

Verweis des Zeigers `p` auf ein Array

Der Einfachheit halber sind nur zehn Elemente in dem Array dargestelt.

Wie kommt man jedoch auf das zweite Element in dem Array, wenn man nicht mit der direkten Zuweisung `p = &buffer[1];` arbeiten, sondern relativ von der ersten auf die zweite Position gelangen möchte? Die Elemente haben die Größe des Typs `short`: 2 Bytes. Der Prozessor muss also für diese Operation eine Addition des Adresswertes mit 2 ausführen. Der folgende Code beinhaltet bereits diese Addition mit der Größe des zugrunde liegenden Datentyps:

Listing 2.20. Array-Syntax

```
short buffer[100];
short *p;

p = &buffer[0];
p++;
/* p zeigt jetzt auf buffer[1] */
```

Auch dazu gibt es eine Abbildung:

Verweis des Zeigers `p` auf das zweite Element des Arrays

Der Inkrementoperator inkrementiert den Zeiger auf das nächste Element. Dazu muss er die Größe der Elemente kennen, damit er die richtige Operation auf Maschinenebene auslösen kann. Nicht nur der Inkrementoperator ist auf Zeiger anwendbar:

Listing 2.21. Array-Syntax

```
short buffer[100];
short *p;

p = &buffer[2];
p += 5;
/* p zeigt jetzt auf buffer[7] */
```

Im Listing wird der Zeiger `p` um 5 erhöht. Das heißt, dass er um fünf Positionen verschoben wird.

Es sind beliebige Additionen oder Subtraktionen ganzzahliger Werte auf einen Zeigerwert anwendbar. Dabei wird ein fremd typisiertes (ganzzahliges) Element innerhalb einer Operation in den Zeigertyp konvertiert[21]. Im vorliegenden Fall ist es die Konstante 5, die eigentlich einen `int`-Wert repräsentiert. Sie wird mit der Addition in den Typ `short *` konvertiert (was intern einer Multiplikation des `int`-Wertes mit der Größe von `short`, also 2, entspricht). Diese Fähigkeit zur Zeigerarithmetik kann man in C und C++ zur Implementierung sehr schneller Speicherzugriffe nutzen.

Listing 2.22. Zeigerinkrementierung auf einem Array

```
short buffer[100];
int i;
short *p;
// ...
p = buffer;
for( i = 0; i < 100; ++i )
  *(p++) = 0;
```

Der Beispielcode setzt alle Elemente des Arrays auf den Wert `0`. Der Zeiger `p` wird dafür auf den Anfang des Arrays gerichtet. Dazu muss nicht unbedingt das erste Element mit dem Klammeroperator bestimmt werden. Der Bezeichner des Arrays stellt schon einen Zeiger auf dessen Anfang (also auf dessen erstes Element) dar. Allerdings ist der Bezeichner des Arrays ein konstanter Zeiger, das heißt ein Zeiger, der selbst nicht geändert werden kann.

Das letzte Beispiel kann auch folgendermaßen umgeschrieben werden, sodass keine explizite Zeigerarithmetik stattfindet:

[21] Siehe auch Abschnitt 2.2.12 auf Seite 39 und Abschnitt 2.2.11 auf Seite 37 zu diesem Thema.

Listing 2.23. Indizierung eines Arrays über einen Zeiger

```
short buffer[100];
int i;
// ...
for( i = 0; i < 100; ++i )
  buffer[i] = 0;
```

Dieser Weg ist für diese einfache Situation möglicherweise der direkteste. Zum Zwecke der Anschaulichkeit soll das Beispiel noch einmal umformuliert werden:

Listing 2.24. Zugriff auf ein Array mittels Zeigerarithmetik

```
short buffer[100];
int i;
// ...
for( i = 0; i < 100; ++i )
  *(buffer + i) = 0;
```

Dass `buffer` ein Zeiger auf den Anfang eines Arrays ist, haben wir weiter oben schon kennen gelernt. Durch die Addition mit einem ganzzahligen Wert errechnet man eine neue Position im Speicher, auf die dann schließlich mit dem Dereferenzierungsoperator zugegriffen wird. Der Klammeroperator `[]` des vorigen Beispiels ist also nichts anderes als das Zusammenspiel einer Addition eines Zeigers mit der nachfolgenden Dereferenzierung.

An späterer Stelle wird noch häufiger auf Zeiger eingegangen werden. Insbesondere im Zusammenhang mit Klassen, Strukturen und Funktionen ist der Einsatz von Zeigern von großer Bedeutung.

2.2.10 Referenzen

Die Referenzen sind eine Art von Datentypen, die C++ gegenüber C neu aufgenommen hat. Es handelt sich bei den Referenzen um Indirektionen, wie bei den Zeigern auch. Es sind jedoch Indirektionen, die während ihres Bestehens nicht mehr verändert werden können. Sie verweisen ab ihrer Definition immer auf das gleiche Objekt und damit auf die gleiche Speicherstelle. Dabei verhalten sie sich syntaktisch nicht wie Zeiger, sondern wie die Objekte, auf die sie verweisen. Eine hinreichende Notwendigkeit zur Einführung der Referenzen in C++ kann an dieser Stelle noch nicht verdeutlicht werden. Dazu müssen erst die Techniken im Zusammenhang mit der Klasseninitialisierung und der Operatorüberladung besprochen werden. Soviel kann jedoch gesagt werden: ohne die syntaktische Form der Referenzen ließen sich manche C++-Konstrukte nicht verwirklichen. Aber nun zu der Syntax der Referenzen selbst:

Listing 2.25. Referenzsyntax

```
int i;
int &r = i;
// ...
r = 7;
printf( "‘%d\n"', i );
```

Die Variable `r` ist im Beispiel eine Referenzvariable. Sie verweist auf `i`, hat aber keine Zeigersyntax, sondern verhält sich so, als wäre sie das Objekt selbst. Wie in dem Beispiel gezeigt, muss die Referenzvariable an der Stelle der Definition auch initialisiert werden. Die Initialisierung einer Referenz bedeutet, dass sie auf eine Speicherstelle gerichtet wird. Danach kann die Referenz nicht mehr auf eine andere Speicherstelle umgerichtet werden.

Technisch macht der Compiler aus der Referenz das, was er auch aus einem Zeiger macht. Zeiger und Referenzen verhalten sich also aus Laufzeitgesichtspunkten heraus identisch. Dennoch tendiert man in C++ an vielen Stellen eher zur Verwendung von Referenzen als von Zeigern.

2.2.11 Typenkonvertierung

In C und C++ sind Daten typisiert. Das heißt, dass alle Daten mit einer Information ausgestattet sind, wie sie im Speicher liegen und wie die Bytes dieses Speichers interpretiert werden. Das hat unter anderem mathematische Konsequenzen für Zahlenwerte, wie sie im Abschnitt 2.2.4 auf Seite 22 beschrieben werden. Der Datentyp ist also die Form, in der die Daten vorliegen. Sollen Daten mit unterschiedlichem Datentyp miteinander über eine Operation in Beziehung gebracht werden, so muss eine Typenkonvertierung erfolgen.

Eine einfache Typenkonvertierung liegt beispielsweise vor, wenn ein ganzzahliger Wert einer Fließkommavariablen zugewiesen wird.

Listing 2.26. Implizite Typenkonvertierung

```
long i;
double d;
// ...
i = 42;
d = i;
```

In diesem Fall müssen Daten, die vorher in vier Bytes Platz hatten, in die acht Bytes der `double`-Variablen verteilt werden – natürlich streng nach den dafür vorgesehenen Regeln. Im Beispiel wird diese Konvertierung durch den Compiler erledigt, da es keine mathematischen oder sonstigen Gründe gegen eine solche Konvertierung gibt. Der Wertebereich des `long`-Datentyps wird durch den Wertebereich des `double`-Datentyps umschlossen. Wenn jedoch

der Wertebereich des Zieltyps kleiner ist als der des Ursprungstyps, dann muss eine so genannte „explizite Konvertierung“ durchgeführt werden. Damit macht man klar, dass man sich der Tatsache, dass Werte verloren gehen können, bewusst ist.

Listing 2.27. Explizite Typenkonvertierung

```
long i;
short s;
// ...
i = 42;
/* Typenkonvertierung */
s = (short)i;
```

Die in dem Beispiel gezeigte Konvertierung wurde nach der alten C-Syntax durchgeführt. Man schreibt vor den Ausdruck, den man konvertieren möchte, den Zieltyp in Klammern. Die Konvertierung `(short)i` kann man in C++ auch so schreiben: `short(i)`. Man nimmt nicht den Zieltyp in die Klammern, sondern den zu konvertierenden Ausdruck. Man nennt diese C++-typische Schreibweise auch die Konstruktorschreibweise. In dem Abschnitt 2.2.22 auf Seite 91 ff. wird dieser Begriff genauer erklärt. In dem vorliegenden Abschnitt soll erst einmal auf die Schreibweise eingegangen werden. Eine weitere C++-typische Variante der Typenkonvertierung zwischen Standardtypen ist die Verwendung des Typenkonvertierungsoperators `static_cast<>()`. Dieser Operator kann nur in bestimmten Fällen zur Typenkonvertierung eingesetzt werden. Die Konvertierungen zwischen den Standardtypen in C++ sind solche Fälle. Die oben beschriebene Konvertierung lässt sich also auch folgendermaßen ausdrücken: `s = static_cast<short>(i);`.

Insbesondere der Operator `static_cast<>()` steht in einem Zusammenhang, der erst im Zusammenhang mit dem Klassenkonzept und den Konstruktoren erklärbar wird. Auch er wird im Abschnitt 2.2.22 näher erklärt. Auf einen weiteren Bereich der Typenkonvertierung soll aber an dieser Stelle noch eingegangen werden: die Konvertierung zwischen Zeigertypen.

Wie schon erwähnt wurde, sind die Zeiger in C und C++ im Allgemeinen typisiert. Das heißt, dass sie auf den Inhaltstyp des Speichers geeicht sind, auf den sie verweisen können. Welchen Sinn kann es haben, Zeigertypen zu konvertieren? Im Allgemeinen ergibt es keinen Sinn. Es gibt allerdings die Ausnahme, dass man einen untypisierten Speicherbereich angefordert hat, den man mit einem bestimmten Datentyp beschreiben möchte. Es gibt beispielsweise Funktionen in der Standardbibliothek, die Speicher auf dem Heap allokieren. Eine solche Funktion ist zum Beispiel `malloc()`. Diese liefert einen untypisierten Zeiger auf diesen Speicher zurück. Ein solcher Zeiger hat den Typ `void *` (mit diesem Datentyp kann man ihn auch anlegen, `void *p;`). Wenn man auf den Speicher nun lesend und schreibend zugreifen möchte, ist eine Typisierung erforderlich, denn in der Typisierung steckt das Konzept,

nach dem der Speicherinhalt interpretiert werden soll. Man kann also auch sinnvoll einen untypisierten Zeiger in einen typisierten wandeln.

Listing 2.28. Dynamische Speicherallokation mit Hilfe von Zeigern

```
void *p;
int *ip;
/* Speicheranforderung... */
p = malloc( 100 * sizeof(int) );
ip = (int *)p;
/* ... */
```

Die Konvertierung des Zeigers funktioniert nach dem gleichen Prinzip wie die eines Wertes. Der Zieltyp muss in Klammern geschrieben werden. Da es sich hierbei um einen Zeigertyp handelt, gehört der Stern `*` zur Typenbeschreibung.

Ein typisierter Zeiger ist implizit in einen untypisierten konvertierbar (`void *`). Natürlich kann jede implizite Typenkonvertierung auch explizit ausgedrückt werden.

2.2.12 Ausdrücke

Der Begriff „Ausdruck" kann von zwei Seiten betrachtet werden. Erstens ist er eine Schreibweise für einen beliebigen Wert. Es handelt sich bei einem Ausdruck nicht um den Wert selbst, sondern um eine Beschreibung desselben mit Hilfe eines oder mehrerer anderer definierter Elemente. Aus diesem Grund kann ein Ausdruck zweitens auch als Verarbeitungsvorschrift definiert werden. In dieser Vorschrift – also im Ausdruck – ist definiert, wie über deren Teile ein Wert bestimmt werden soll.

Im C++ ISO Standard 14882 ist „Ausdruck" wie folgt definiert: *Ausdrücke sind eine Folge von Operatoren und Operanden, die eine Berechnung bewirken. Ausdrücke können einen Wert ergeben und können Nebenwirkungen (Seiteneffekte) hervorrufen.*[22]

In Ausdrücken in C und C++ kann man prinzipiell zwei Dinge errechnen: erstens einen Wert, und zweitens eine Adresse. Um zu verdeutlichen, was damit gesagt ist, kann die folgende Zuweisung betrachtet werden:

```
*(werte + i%100) = m * 10;
```

Auf der linken Seite der Zuweisung wird eine Speicherstelle zur Aufnahme eines Wertes bestimmt. Wahrscheinlich handelt es sich bei `werte` um ein Array oder eben um einen Zeiger auf den Anfang eines Speicherbereichs. Die Addition und die Modulooperation, die innerhalb des Ausdrucks auf

[22] C++ ISO Standard 14882 Kapitel 5 Absatz 2.

der linken Seite erfolgen, sind Adressoperationen. Auf der rechten Seite wird einfach ein Wert berechnet, der dann durch die Zuweisung auf die linke Seite transportiert wird. Dort findet die Zuweisung einen Platz für den Wert. Ausdrücke, die Adressen berechnen, können auch innerhalb von Ausdrücken stehen, die Werte berechnen, und anders herum. Die linke Seite einer Zuweisung bestimmt allerdings immer eine Speicherstelle und wird „L-Wert“ oder „L-Value“ genannt. Die rechte Seite einer Zuweisung bestimmt immer einen Wert im Sinne der Typisierung des Ausdrucks[23] und wird als „R-Wert“ oder „R-Value“ bezeichnet.

Wenn die Teile eines Ausdrucks durch Operationen miteinander in Verbindung gebracht werden, müssen deren Typen zueinander passen. Haben Teilausdrücke unterschiedliche Typen, so entsteht die Notwendigkeit zur Typenkonvertierung. Im vorangegangenen Abschnitt wurden die einfachen Typenkonvertierungen besprochen. Es gibt Konvertierungen, die vom Compiler implizit durchgeführt werden können. Das sind all jene, die von der Betrachtung des Wertebereichs aus gesehen erweiternde Konvertierungen sind. Außerdem können ganzzahlige Werte implizit in Zeigertypen konvertiert werden (ausgenommen `void *`), um L-Werte zu berechnen. Wenn eine notwendige Typenkonvertierung nicht durchgeführt werden kann, dann muss der Compiler eine Fehlermeldung ausgeben und den Entwickler darauf hinweisen, dass er eine Konvertierung einfügen muss.

Wie das genau funktioniert, soll hier an einem Beispiel verdeutlicht werden:

Listing 2.29. Implizite Typenkonvertierung in einem Ausdruck

```
int i = 5;
double d = 7.3;
double e;
/* Implizite Konvertierung des Wertes i nach double */
e = i + d;
```

2.2.13 Operatoren

C++ stellt Operatoren für verschiedene Zwecke zur Verfügung. Da gibt es die arithmetischen Operatoren zur Formulierung von Rechenanweisungen. Eine ähnliche Operatorenfamilie ist die der Bitoperatoren, die Berechnungen auf Bit-Ebene ermöglicht. Es existiert ein Zuweisungsoperator, der häufig mit den arithmetischen Operatoren gemeinsam in einer Anweisung verwendet wird. Etwas besonders im Kanon der Programmiersprachen ist in C und C++

[23] Das kann auch wieder ein Adresswert sein. In diesem Fall bestimmt der L-Wert eine Speicherstelle, in die ein Adresswert passt.

das Vorhandensein kombinierter Operatoren für Arithmetik und Zuweisung. Zur Formulierung boolscher Ausdrücke gibt es verschiedene Vergleichsoperatoren und die so genannten logischen Operatoren. C- und C++-typische Inkrement- und Dekrementoperatoren bieten eine weitere Möglichkeit Zahlen zu manipulieren.

Arithmetische Operatoren

In den vorangehenden Abschnitten habe ich schon einige arithmetische Operatoren verwendet, ohne sie explizit einzuführen. In diesem Abschnitt sollen alle arithmetischen Operatoren aus C++ aufgelistet werden.

Arithmetische Operatoren können innerhalb von Ausdrücken verwendet werden, die einen Wert eines beliebigen Typs berechnen. Folgende arithmetische Operatoren existieren in C++:

`+` Addition
`-` Subtraktion
`*` Multiplikation
`/` Division
`%` Modulo (Restwert einer Division)

Das Ergebnis einer Operation hat immer den Typ der Operanden. Was bei Additionen und Subtraktionen intuitiv erscheint, lohnt bei einer Division eine Betrachtung. So ist das Ergebnis von `23 / 10` eine ganzrationale `2`. Wünscht man ein gebrochenrationales Ergebnis, muss mindestens ein Operand einen Fließkommatyp haben. In diesem Fall wird auch eine Fließkommaoperation durchgeführt. Der Ausdruck `23.0 / 10.0` hat `2.3` als Ergebnis. Die Moduloperation kann nur auf ganzzahlige Operanden durchgeführt werden, denn sie ermittelt den Restwert einer Division bei ganzzahligem Ergebnis.

Unäre Vorzeichenoperatoren

Für einen Vorzeichenwechsel kann einem Wert ein Minuszeichen vorangestellt werden. Der Vollständigkeit halber existiert auch der unäre Operator `+`, der keinen Vorzeichenwechsel durchführt.

`+` kein Vorzeichenwechsel
`-` Vorzeichenwechsel

Bitoperatoren

Die Bitoperatoren sind den arithmetischen Operatoren darin verwandt, dass sie syntaktisch genauso angewendet werden wie jene. Ihre Wirkungsweise zielt aber auf die Manipulation von Bitmustern.

`&` bitweises UND
`|` bitweises ODER

^ bitweises exklusives ODER
« bitweises Verschieben nach links
» bitweises Verschieben nach rechts

Der Operator & verbindet zwei Operanden und errechnet das Ergebnis folgendermaßen: Er verrechnet jede einzelne Bitposition der Operanden miteinander und setzt die entsprechende Bitposition im Ergebniswert dann, wenn die beiden Bits der Operanden gesetzt sind. Um ein Ergebnisbit zu setzen muss also das entsprechende Bit im ersten *UND* im zweiten Operanden gesetzt sein. Also:

	1010
&	0110
=	0010

Beim Operator für das bitweise *ODER* | muss das Bit des einen oder des anderen Operanden gesetzt sein, um zu einem gesetzten Ergebnisbit zu führen.

	1010
|	0110
=	1110

Die Operation *EXKLUSIV ODER* ^ setzt das Ergebnisbit dann, wenn nur eines der Operandenbits gesetzt ist.

	1010
^	0110
=	1100

Zu den Bitoperatoren ein kleines Beispiel, das durch Sie leicht auch für alle anderen Bitoperationen modifiziert werden kann:

Listing 2.30. Der Bitoperator &

```
#include <iostream>

int main()
{
  int a = 3;
  int b = 6;
  int c = a & b;
  std::cout << c << std::endl;
  return 0;
}
```

Das Programm in Listing 2.30 gibt den Wert 2 auf der Standardausgabe aus. Wenn man die beiden Werte 3 (binär 11) und 6 (binär 110) miteinander

durch das bitweise UND verrechnet, kommt das Bitmuster `10` (dezimal 2) dabei heraus.

Der Komplementoperator ~ ist ein unärer Operator. Er nimmt nur einen Operanden entgegen, ähnlich dem Vorzeichen-Minus. Das Vorzeichen-Minus dreht das Vorzeichen, der Komplementoperator dreht das Bitmuster. Alle nicht gesetzten Bits werden gesetzt und andersherum. Aus `1001` wird `0110`. Man bringt den Komplementoperator vor dem Operanden an: `~Operand`.

~ Komplement eines Bitmusters

Zuweisungsoperatoren

Der Zuweisungsoperator transportiert den auf der rechten Seite errechneten Wert in eine auf der linken Seite angegebene Speicherstelle. Man spricht deshalb auch von *R-Values*, wenn es um eine Werteberechnung geht, und von *L-Values*, wenn es um die Bestimmung von Adresswerten zur Speicherung von Daten geht.

= Zuweisung

Eine Besonderheit in C und C++ ist, dass es einen Satz von Operatoren gibt, die die Berechnung und die Zuweisung zusammenziehen. Wenn also eine solche Operation angewendet wird, wird der linke Operand dabei verändert. So hat beispielsweise der Ausdruck `x += 5;` eine Erhöhung von `x` um den Wert `5` zur Folge.

+= Zuweisung in Verbindung mit Addition
-= Zuweisung in Verbindung mit Subtraktion
*= Zuweisung in Verbindung mit Multiplikation
/= Zuweisung in Verbindung mit Division
%= Zuweisung in Verbindung mit Modulooperation
&= Zuweisung in Verbindung mit bitweisem UND
|= Zuweisung in Verbindung mit bitweisem ODER
≙ Zuweisung in Verbindung mit bitweisem exklusiv ODER

Vergleichsoperatoren

Die Vergleichsoperatoren haben einen boolschen Ergebniswert.

== Prüfung auf Gleichheit
!= Prüfung auf Ungleichheit
< kleiner
> größer
<= kleiner gleich
>= größer gleich

Logische Operatoren

Mit den logischen Operatoren werden Teilbedingungen zusammengefasst. Es werden also boolsche Werte miteinander in Beziehung gesetzt und daraus ein boolsches Ergebnis berechnet.

`&&` logisches UND
`||` logisches ODER

`!` NOT-Operator, Umkehrung eines logischen Ausdrucks

Neben diesen Operatoren, die alle etwas damit zu tun haben irgendwelche Werte zu bestimmen oder zu verändern, gibt es noch eine ganz andere Art von Operatoren, die damit zu tun haben auf Daten zuzugreifen.

`[]` Index-Operator zur Indizierung von Arrays
`.` Punktoperator zum Zugriff auf Struktur- und Klassenelemente
`->` Pfeiloperator zum Zugriff auf Struktur- und Klassenelemente über einen Zeiger
`&` Adressoperator, ermittelt Adresse eines Objekts
`*` Dereferenzoperator, gibt den Inhalt einer Speicherstelle
`::` Scope-Operator, ordnet Elemente einer Struktur oder einer Klasse zu

`?:` Bedingungsoperator, mit dem man bedingte Ausdrücke oder bedingte Anweisungen formulieren kann

2.2.14 Anweisungen

Mit Anweisung ist jede sprachliche Arbeitsvorschrift gemeint. Anweisungen werden in C und C++ mit einem Semikolon abgeschlossen. Die Deklaration einer Konstanten oder einer Variablen ist zum Beispiel keine Anweisung, weil keine Arbeit zur Laufzeit des Programms dadurch anfällt.

Listing 2.31. Deklaration und Anweisungen

```
int a, b;      /* Deklarationen */
a = 5;             /* Anweisung */
b = a + 3;         /* Anweisung */
funktion( a, b ); /* Anweisung */
```

In der Zeile 1 des Listings werden zwei Variablen deklariert. Das ist keine Arbeitsvorschrift, weshalb man es auch nicht als Anweisung bezeichnet. In den folgenden zwei Zeilen werden den Variablen `a` und `b` Werte zugewiesen. Die Zuweisungen und die Addition mit dem Wert 3 wird zur Laufzeit durchgeführt. Die CPU braucht dafür etwas Zeit. Man spricht daher bei diesen Zeilen von Anweisungen.

2.2.15 Kontrollstrukturen

C++ hat die Kontrollstrukturen von C übernommen und keine weiteren hinzugewonnen. Die in diesem Abschnitt gezeigten Strukturen sind also für beide Sprachen gleichermaßen gültig.

Es gibt genau eine if-else-Verzweigung, drei Schleifentypen und eine Verzweigung nach einer Sprungtabelle. Der if-else-Konstruktion und den drei Schleifentypen ist gemeinsam, dass sie Bedingungen enthalten, die zur Laufzeit abgeprüft werden. Aus diesem Grund soll zuerst auf die Bedingung bzw. auf den Bedingungsausdruck in C++ eingegangen werden.

Bedingungen

In C ist jeder Wert ungleich 0 gleichbedeutend mit *wahr*. Die 0 stellt *unwahr* dar. In C gibt es dafür keine Schlüsselworte, die genannte Definition reicht vollkommen aus. In C++ hat man mit dem Standard von 1998 die Schlüsselworte `true` und `false` eingeführt, die mit 1 und 0 definiert sind. Für das Ergebnis einer Bedingung wurde auch der Typ `bool` eingeführt, der die beiden Werte `true` und `false` annehmen kann. Trotzdem gilt die alte Definition aus C weiter, die jeden anderen Wert außer 0 als Wahrheitswert betrachtet. In einer `if`-Abfrage kann also jeder beliebige mathematische oder bitarithmetische Ausdruck stehen, der einen Standarddatentyp zum Ergebnis hat. Neben den mathematischen Ausdrücken gibt es solche, die überprüfender und vergleichender Natur sind. Dafür gibt es extra Vergleichsoperatoren, die in der Lage sind Werte zu vergleichen und einen boolschen Wert als Ergebnis zu liefern. Der Vergleichsoperator für den direkten Vergleich in C/C++ ist `==`. Der Ausdruck `x == 5` liefert entweder `true` oder `false`. Insgesamt gibt es die folgenden Vergleichsoperatoren:

```
==   gleich
!=   ungleich
<    kleiner
>    größer
<=   kleiner gleich
>=   größer gleich
```

Neben den Vergleichsoperatoren gibt es noch solche, die es erlauben einen Bedingungsausdruck aus mehreren Teilen zusammenzusetzen. Verknüpft man zwei Teilausdrücke mit `&&`, so ist es notwendig, dass beide Teilausdrücke wahr sind, damit der gesamte Ausdruck wahr sein kann. Also `A == 5 && B != 7` ist ein Bedingungsausdruck, der wahr ist, wenn A den Wert 5 besitzt und zusätzlich B nicht mit 7 belegt ist. Der Operator `||` verbindet die Teilausdrücke mit *oder*. Für den Ausdruck `A == 5 || B != 7` gilt, dass der Gesamtausdruck wahr ist, wenn einer der beiden Teilausdrücke zutrifft. Eine Bedingung kann

aus beliebig vielen Teilbedingungen zusammengesetzt werden, wobei auch geklammert werden kann. Um eine Bedingung oder eine Teilbedingung zu negieren, gibt es den Operator `!`.

`if()-else`

Die Kontrollstruktur `if()-else` kann die folgenden vier Ausprägungen haben:

```
if( Bedingung )
  Anweisung;
```

oder

```
if( Bedingung )
{
  Anweisung1;
  Anweisung2;
  ...
}
```

oder

```
if( Bedingung )
  Anweisung1;
else
  Anweisung2;
```

oder

```
if( Bedingung )
{
  Anweisung1;
  Anweisung2;
  ...
}
else
{
  Anweisung3;
  Anweisung4;
  ...
}
```

`while()`

Die `while()`-Schleife kann so

```
while( Bedingung )
  Anweisung;
```

oder so

```
while( Bedingung )
{
  Anweisung1;
  Anweisung2;
  ...
}
```

aussehen.

do - while()

Die `do - while()`-Schleife ist die einzige fußgesteuerte Schleife in C++. In ihr steht die Bedingung im Schleifenfuß, weshalb sie mindestens einmal durchlaufen wird.

```
do
  Anweisung;
while( Bedingung );
```

Oder

```
do
{
  Anweisung1;
  Anweisung2;
  ...
} while( Bedingung );
```

for()

Wie die `while()`-Schleife ist die `for()`-Schleife eine durch den Schleifenkopf gesteuerte Schleife. Das heißt, dass sie, wenn die Bedingung von Anfang an nicht zutrifft, kein einziges Mal durchlaufen wird. Die `for()`-Schleife kann durch eine `while()`-Schleife ersetzt werden. Sie ist etwas aufwändiger als die `while()`-Schleife. Trotzdem ist sie die Schleifenform, die am häufigsten in C- und C++-Quellcodes zu finden ist. Das liegt daran, dass sie für ein ganz besonderes Problem entworfen wurde, das in der Programmierung ständig wiederkehrt. Es ist das Durchzählen einer Indexvariablen zwischen zwei Grenzen mit einer bestimmten Schrittweite. Dabei findet man alles, was zu dieser Indexzählung gehört, im Schleifenkopf der `for()`-Schleife:

```
for( Initialanweisung; Bedingung; Schleifenendanweisung )
{
}
```

In der Initialanweisung wird üblicherweise der Index deklariert und initialisiert. Häufig existiert die Indexvariable schon und wird nur initialisiert. Natürlich kann auch jede andere Anweisung an der Stelle der Initialanweisung der `for()`-Schleife ausgeführt werden. Die Bedingung definiert die Regel, die zu erneuten Schlefendurchläufen führt. Meistens kontrolliert sie den Wertebereich des Indexes. Die Schleifenendanweisung wird nach dem Schleifenblock ausgeführt und wird meistens dazu verwendet die Indexvariable um einen Betrag x zu verändern.

Das folgende Codebeispiel ist eine sehr übliche Form der Anwendung einer `for()`-Schleife. Sie durchläuft den Schleifenrumpf zehnmal für die Indexwerte `0` bis `9`.

```
for( int i = 0; i < 10; ++i )
{
 // Anweisungen...
}
```

`switch()-case`

Die Switch-Case-Kontrollstruktur entscheidet anhand eines Wertes x, zu welchem Label im Case-Block ein Sprung ausgeführt werden soll. Der Wert wird der `switch`-Anweisung übergeben. Innerhalb des Blocks stehen die `case`-Labels mit den dazugehörigen Werten, die als Einsprungmarken dienen. Die Werte hinter dem Label steuern den Programmfluss, indem sie bei passendem Wert in Switch den Beginn der Programmfortführung definieren. Ein solches `case`-Label definiert aber kein Ende eines vorangegangenen Blocks. Ganz im Gegensatz zu manchen ähnlichen Verzweigungsstrukturen in anderen Programmiersprachen bedeutet der nächste `case`-Block nicht das Ende des vorangegangenen. Das ist wichtig zu beachten, denn es stellt eine häufige Fehlerquelle dar. Wenn man einen Block beenden möchte, so kann man das mit der Anweisung `break;` tun. Bereiche lassen sich nach `case` nicht angeben. Dafür können mehrere `case`-Labels nacheinander aufgeführt werden.

```
switch( x )
{
  case 1:
    Anweisung1;
    break;
  case 2:
    Anweisung2:
    break;
  case 3:
  case 4:
    Anweisung3;
    break;
```

```
  default:
    Anweisung4;
    break;
}
```

Für alle Werte, die nicht gesondert mit `case`-Labels ausgestattet sind, kann das Label `default` eingesetzt werden. Die Reihenfolge der `case`-Labels innerhalb eines `switch`-Blocks ist dabei gleichgültig. Auch `default` darf an jeder beliebigen Position stehen. Wichtig ist nur, dass ein Labelwert nur einmal auftaucht.

2.2.16 Funktionen

Funktionen sind in C++, wie in C auch, separate Blöcke mit ausführbarem Code. Funktionen können zu ihrer Ausführung Daten übergeben bekommen. Dazu haben sie eine Parameterliste, die genau beschreibt, welche Datentypen entgegengenommen werden können und in welcher Reihenfolge diese übergeben werden müssen. Nach der Ausführung können Funktionen auch Daten zurückliefern. Dazu gibt es die Deklaration eines Rückgabewertes, die auch den Datentyp festlegt. Wenn mehrere verschiedene Daten durch eine Funktion produziert werden, gibt es die Möglichkeit, einen zusammengesetzten Typ – eine Struktur, wie sie in Abschnitt 2.2.21 beschrieben ist – zurückzuliefern, oder mit Zeigern auf Daten zu arbeiten. Eine Funktion kann auch ohne Parameter und ohne Rückgabewert auskommen. Die Deklaration einer Funktion folgt dem Schema:

```
TYP1 Funktionsname( TYP2 Parameter1, TYP3 Parameter2 );
```

Die Parameter einer Funktion werden mit Komma getrennt. Funktionen können Klassenelemente sein. Man nennt sie dann Methoden[24]. Jeder ausführbare Code in einem Programm befindet sich innerhalb von Funktionen[25]. Die Funktion hat einen Funktionsblock, der die Anweisungen enthält. Dieser Block wird durch geschweifte Klammern zusammengefasst.

```
TYP1 Funktionsname( TYP2 Parameter1, TYP3 Parameter2 )
{
Anweisung1;
Anweisung2;
...
}
```

[24] Siehe Abschnitt 2.2.21 auf Seite 73.
[25] Von Initialisierungsanweisungen auf dem globalen Scope einmal abgesehen.

Der Aufruf einer Funktion findet durch die Angabe des Namens statt. Dem Namen der Funktion folgt die Übergabe der Parameter in einfachen Klammern. Die Parameter werden wieder durch Komma getrennt. Da die Daten, die übergeben werden, typisiert sind, muss beim Funktionsaufruf kein Typ angegeben werden. Wenn die Funktion einen Wert zurückliefert, wird dieser häufig in Form einer Zuweisung an eine Variable auf dem aktuellen Scope abgeholt. Eine Funktion kann aber auch Teil eines Ausdrucks sein oder in einer Parameterliste einer anderen Funktion stehen. Der allgemeine Startpunkt für ein Programm ist die Funktion `main()`.

Listing 2.32. Einfache Funktionen

```
#include <iostream>

int f1( int i )
{
  return i * 2;
}

int f2( int i )
{
  return i * i;
}

int main()
{
  int i;
  i = f2( f1( 5 ) );
  std::cout << i << std::endl;
  return 0;
}
```

Dieses Programm startet mit der Ausführung der Funktion `main()`. Innerhalb von `main()` wird eine Variable `i` deklariert, die den Typ `int` besitzt. Die Zeile nach der Deklaration von `i` ist bereits eine Anweisung. Die Anweisung besteht aus den verschachtelten Aufrufen der Funktionen `f1()` und `f2()` sowie einer Zuweisung an `i`. Dabei wird zuerst die Funktion `f1()` aufgerufen und ihr der Wert 5 übergeben. Die Definition der Funktion weist eine Parameterliste mit einem Parameter des Typs `int` aus. Da 5 ein ganzzahliger Wert innerhalb des Wertebereichs von `int` ist, kann der Parameter `i` der Funktion diesen Wert aufnehmen. Wohlgemerkt: der Parameter `i` hat mit dem `i` der Funktion `main()` nichts zu tun. Er könnte ebenso einen beliebigen anderen Bezeichner tragen. Man könnte die Funktion also folgendermaßen definieren:

Listing 2.33. Einfache Funktion

```
int f1( int xyz )
{
  return xyz * 2;
}
```

Es würde sich nichts an deren Funktionalität ändern. Das Programm würde ebenso ablaufen, wie es mit dem Parameterbezeichner `i` schon tut. Die Funktion `f1()` errechnet einen Rückgabewert, den sie auch mit ihrem Ablauf zurückliefert. Dieser Wert wird durch die Funktion `f2()` als Parameter übernommen. Auch hier ist es die Typisierung, die die Parameterübergabe passend macht. und natürlich spielt auch hier die Benennung des Parameters der Funktion keine Rolle. Er ist unabhängig von allem, was sonst noch im Programm definiert wurde. Dass der Name `i` gewählt wurde, bringt ihn nicht in Konflikt mit anderen Variablen, die in anderen Gültigkeitsbereichen definiert wurden. Auch die Funktion `f2()` liefert einen Wert zurück. Dieser wird nun durch eine Zuweisung an die Variable `i` innerhalb von `main()` übergeben.

Die Deklaration einer Funktion muss zu ihrer Definition passen. Das heißt, dass der Rückgabetyp, der vor dem Namen der Funktion steht, und die Parameterliste nach dem Funktionsnamen festlegen, was innerhalb der Funktion getan werden kann. Die Returnanweisung am Ende der Funktion muss zum Beispiel den richtigen Typ zurückliefern. Tut sie es nicht, mahnt der Compiler einen Fehler an. Ebenso kann mit den Parametern in einer Funktion nur in der Art verfahren werden, wie sie durch die Datentypen vorgegeben ist. Wenn ein Parameter als `int` deklariert ist, bekommt man mit ihm keine Fließkommazahlen transportiert. Man müsste dazu einen anderen Parametertyp wählen.

Der Prototyp

Die Deklaration einer Funktion kann von deren Definition abgetrennt werden. Das ist insbesondere dann wichtig, wenn die Funktion in einer anderen Übersetzungseinheit (Modul) stehen soll, als der Aufruf der Funktion. Meistens gibt es für Funktionen viele Aufrufe, aber natürlich nur eine Definition. In diesem Fall also muss der aufrufenden Stelle die Deklaration einer Funktion bekannt gemacht werden, da der Compiler den korrekten Code für deren Aufruf einfügen muss. Eine solche Funktionsdeklaration nennt man *Funktionsprototyp* oder einfach *Prototyp*.

Ein solcher Prototyp sieht wie die erste Zeile der Funktionsdefinition aus. Es wird der Rückgabetyp vor dem Namen angegeben, der von der Parameterliste gefolgt wird. Danach wird ein Funktionsprototyp mit einem Semikolon abgeschlossen. Eine Änderung gegenüber den Funktionsdefinitionen gibt es noch. Die Parameter müssen nicht benannt sein. Es reicht also, wenn nur die Typen der Parameter in der richtigen Reihenfolge durch Komma getrennt

in der Parameterliste stehen. Die Definition der Funktion muss die Deklarationselemente alle noch einmal wiederholen. Außerdem müssen jetzt die Parameter benannt werden, wenn man sie benutzen möchte[26].

Der Prototyp für die Funktion

Listing 2.34. Funktionsdefinition

```
int f1( int wert )
{
  return wert * 2;
}
```

sieht also folgendermaßen aus:

Listing 2.35. Funktionsprototyp

```
int f1( int );
```

Er muss keinen Bezeichner in der Parameterliste haben. Wenn ein Bezeichner eingeführt wird, wird er vom Compiler ignoriert. Das heißt, dass es auch ein anderer Bezeichner sein kann als der, der bei der Definition der Funktion verwendet wird.

Listing 2.36. Funktionsprototyp mit benanntem Parameter

```
int f1( int beliebig );
```

Ein Bezeichner in der Parameterliste eines Prototyps hat reinen Kommentarwert.

Üblicherweise werden Funktionsprototypen in Headerdateien aufgenommen. Diese Headerdateien werden in den Modulen inkludiert, in denen die Funktionen aufgerufen werden. Sie werden aber auch in den Modulen inkludiert, in denen die Funktionen definiert werden. Vor allem Letzteres wird manchmal vergessen und kann dann zu Fehlern führen. Denn wenn die Definition einer Funktion nicht mit deren Deklaration übereinstimmt, unterscheiden sich aufrufender und aufgerufener Code derart, dass ein Laufzeitfehler auftritt. Wenn der Funktionsprototyp auch an der Stelle der Definition bekannt gegeben wird – durch das Inkludieren der richtigen Headerdatei –, kann der Compiler einen Unterschied erkennen und einen Fehler liefern. Einen Compilerfehler zu korrigieren ist wesentlich leichter, als einen Laufzeitfehler zu finden.

Listing 2.37. Inklusionsbeziehungen zur Sicherung des passenden Funktionsaufrufs

```
// Datei F.H
int f1( int );
```

[26] Diese Regel gilt für C++. In C gibt es dazu einige Abweichungen, die aber hier nicht näher erörtert werden sollen.

```
// Datei F.CPP
#include "F.H"

int f1( int wert )
{
  return wert * 2;
}

// Datei MAIN.CPP
#include <iostream>
#include "F.H"

int main()
{
  st::cout << f1( 21 ) << std::endl;

  return 0;
}
```

Die Technik der Verwendung von Funktionsprototypen wurde in der Sprache C mit dem ANSI-Standard von 1989 eingeführt. C-Quelltext, der ANSI-konform sein soll, muss sich zwingend an deren Verwendung halten. Aufgerufene Funktionen müssen mit einem Prototyp vordeklariert sein, wenn sie nicht vor der aufrufenden Stelle definiert wurden. Als Faustregel kann man sich merken, dass der Compiler nichts verwenden kann, was nicht vorher definiert oder zumindest deklariert wurde. Der Funktionsprototyp ist in diesem Sinne eine Deklaration. Die Deklaration verweist im Voraus auf die eigentliche Definition der Funktion.

Funktionszeiger

Auch auf Funktionen kann man mit Zeigern zugreifen. Dabei muss der Zeiger mit der exakten Parameterliste und dem Rückgabetyp der Funktion deklariert sein.

Listing 2.38. Ein Funktionszeiger

```
int f( int , double  )
{
  return 0;
}

int (*fp)(int,double); // Funktionszeiger
```

Der Bezeichner einer Funktion stellt ohne die runden Klammern die Adresse der Funktion dar. Man kann also einen Funktionszeiger auf eine

Funktion richten, indem man einfach deren Bezeichner als R-Value verwendet.

```
fp = f;
```

Umgekehrt führen die Klammern zum Aufruf einer Funktion an einer gegebenen Adresse.

Listing 2.39. Aufruf der Funktion über den Funktionszeiger durch die Klammern

```
// Aufruf der Funktion über den Zeiger.
int i = fp(7,3.3);
```

2.2.17 Funktionsüberladung

Während in C ein Bezeichner nur einmal vergeben werden kann, darf man in C++ mehreren Funktionen den gleichen Namen geben. Diese Funktionen müssen sich dann allerdings in der Parameterliste unterscheiden. Beim Aufruf einer Funktion unterscheidet dann der Compiler anhand der übergebenen Datentypen, welche der verschiedenen gleichnamigen Funktionen aufgerufen werden soll. Der Compiler zieht also Name und Parameterliste zur Unterscheidung der Funktionen heran. Diese beiden Kriterien zusammengenommen stellen die sogenannte Signatur einer Funktion dar.

Der Begriff der Signatur

Im Zusammenhang mit Funktionen ist in C++ der Begriff der „Signatur" wichtig. Mit Signatur bezeichnet man den Namen, die Typen der Parameter sowie deren Reihenfolge. In diesen drei Bestandteilen der Signatur kann der Compiler Funktionen voneinander unterscheiden. Der Rückgabetyp zählt nicht zur Signatur.

So sehen beispielsweise verschiedene Signaturen aus. In der Parameterliste werden unterschiedliche Typen verwendet.

```
int minimum( int a, int b );
double minimum( double a, double b );
```

Im Sprachgebrauch der OO-Programmierung hebt man nicht mehr unbedingt hervor, dass es zwei Funktionen `minimum()` gibt. Man sagt einfach, dass die Funktion zwei Überladungen besitzt. Nicht nur unterschiedliche Typen machen unterschiedliche Signaturen aus. Es ist auch die Anzahl an Parametern, die Signaturen unterscheidbar machen.

```
int minimum( int a, int b, int c );
int minimum( int a, int b, int c, int d );
```

Der Rückgabewert wird nicht zur Signatur einer Funktion gerechnet. Er kann zwischen Überladungen zwar variieren, definiert aber keine eigene Überladung. Das heißt, das allein ein Unterschied im Rückgabewert nicht vom Compiler angenommen wird. Der Compiler wird in einem solchen Fall einen Fehler anmahnen.

Listing 2.40. Überladene Funktion `minimum()`

```
// ...

// Prototypen
int minimum( int, int );           // Variante 1
double minimum( double, double );  // V. 2
int minimum( int, int, int );      // V. 3
int minimum( int, int, int, int ); // V. 4

void f()
{
  int a = 1, b = 2, c = 3, d = 4;
  double x = 2.7, y = 7.3;

  std::cout << minimum( d, b ) << std::endl;
  std::cout << minimum( a, b, c, d ) << std::endl;
  std::cout << minimum( x, y ) << std::endl;
  std::cout << minimum( c, x ) << std::endl; // V. 2 !

  // ...
}
```

In dem Beispiel sind die Definitionen – die Implementierungen – der verschiedenen Überladungen von `minimum()` nicht enthalten. Gehen wir einfach davon aus, dass die Implementierungen sich wahrscheinlich in einem anderen Modul befinden. Das ist für die Auswahl der korrekten Funktionen auch ohne Belang. Es wurden ja Prototypen angegeben, die beschreiben, welche Überladungen der Funktion `minimum()` existieren. Der Compiler wählt nun allein anhand der Anzahl und der Typen der Parameter. Die ersten drei Zeilen im Beispiel dürften eindeutig zu interpretieren sein. Was aber ist mit der vierten Zeile? Dort wird ein Parameter mit dem Typ `int` und einer mit `double` übergeben. Der Compiler prüft die möglichen impliziten Konvertierungen der übergebenen Parameter zu den Typen, die in den möglichen Parameterlisten von `minimum()` vorhanden sind. Von `int` auf `double` kann implizit konvertiert werden, ohne Daten zu verlieren. Also konvertiert er den ersten Parameter nach `double`, damit die vorhandene Überladung `minimum(double,double)` aufgerufen werden kann.

Der C++-Compiler versucht alle definierten Wege der Typenkonvertierung[27], um die Anforderungen einer Signatur erfüllen zu können. Definiert sind die Konvertierungen von Standarddatentypen kleiner Wertebereiche hin zu den Typen größerer Wertebereiche. Um folgenden Abschnitten etwas vorzugreifen: Konvertierungen können auch von Standarddatentypen hin zu Klassentypen definiert sein, oder einfach zwischen verschiedenen Klassentypen. Es ist also immer ratsam, sich die möglichen passenden Typen einer Signatur bewusst zu machen.

2.2.18 Funktions - Inlining

Inlining ist ein Thema mit vielen Aspekten, weshalb eigentlich etwas mehr als nur ein eigener Abschnitt für dieses Thema notwendig ist. Es steht mit vielen anderen Bereichen der C++-Programmierung im Zusammenhang. Alle Aspekte des Inlinings von Funktionen werden also erst im Verlauf der folgenden Kapitel behandelt. Zunächst aber die Idee und die syntaktische Umsetzung: eine Inlinefunktion ist eine Funktion, die an der Stelle ihres Aufrufs durch den Compiler durch ihren Inhalt ersetzt wird, ohne eine semantische Änderung durchzuführen. Das heißt, dass im Objektcode kein Funktionsaufruf mehr steht, sondern eine beliebige Abfolge von CPU-Anweisungen, die semantisch der Funktion entsprechen. Dazu gibt es das Schlüsselwort `inline`.

Listing 2.41. Normale Funktion

```
long zinsen( long betrag, double zinssatz )
{
  return (long)(betrag * zinssatz);
}
```

Diese Funktion kann folgendermaßen zu einer Inlinefunktion gemacht werden:

Listing 2.42. Inlinefunktion

```
inline long zinsen( long betrag, double zinssatz )
{
  return (long)(betrag * zinssatz);
}
```

und kann damit vom Compiler optimiert werden.

Allerdings ist der `inline`-Modifizierer nur eine Empfehlung an den Compiler. Das bedeutet, dass der Compiler nicht optimieren muss. Es gibt nun einige Compiler, die sehr gut und grundsätzlich alle Inlinefunktionen optimieren, wenn es technisch möglich ist. Andere Compiler tun dies nur auf

[27] Im Englischen nennt man die Typenkonvertierung „Typecast“ oder verkürzt einfach „Cast“. In einigen Abschnitten des Buchs wird die englische Benennung verwendet.

einen expliziten Verweis in Form eines Compilerschalters. Eine dritte Gruppe von Compilern hat generelle Probleme mit dem Inlining und kann nur ganz bestimmte Funktionen optimieren. Damit ist schon eine Krux angesprochen, die das Funktionsinlining mit sich bringt. Man kann sich nur bedingt darauf verlassen.

Ungeachtet der Tatsache, dass es auch technische Gründe geben kann, die eine Inline-Expansion von Funktionen unmöglich machen – an späterer Stelle wird darauf eingegangen werden –, befassen wir uns hier mit den Gründen für das Funktionsinlining. Ein traditioneller Grund für das Inlining ist die reine Performanzsteigerung des Codes. Wenn der Compiler das Inlining durchführt, wird zumindest ein Funktionsaufruf gespart. Dabei sollte aber beachtet werden, dass wenn die Funktion einen großen Block hat, eine unverhältnismäßige Codeaufblähung stattfinden kann. Bedenkt man, dass Ausführungsgeschwindigkeit nicht nur durch Reduktion der abzuarbeitenden CPU-Anweisungen erreicht werden kann, sondern auch durch Reduktion des gesamten ausführbaren Codes, kann Inlining auch kontraproduktiv sein[28]. Moderne Prozessoren arbeiten Funktionsaufrufe sehr performant ab, was in den meisten Fällen ein Inlining aus Performanzgründen überflüssig macht. Allerdings können sehr kleine Funktionen sehr gut optimiert werden. Insbesondere dann, wenn die in der Funktion enthaltenen Anweisungen weniger aufwändig sind als der Funktionsaufruf selbst. In diesen Fällen kann eine Steigerung der Performanz und eine Verkleinerung des Codes erreicht werden. Eine Technik, die häufig im Zusammenhang mit Klassenmethoden angewandt wird.

Ein Problembereich bleibt allerdings. Im Allgemeinen werden Inlinefunktionen in Headerdateien definiert, damit sie durch den Compiler an den Stellen optimiert werden können, wo sie aufgerufen werden. Wenn nun der Compiler nicht in der Lage ist, eine Ersetzung durchzuführen, oder es aus irgendeinem anderen Grund nicht tut, muss die Funktion als reguläre Funktion in die Objektdatei aufgenommen werden. Da nun Headerdateien normalerweise öfter als nur einmal in Module eines Projekts inkludiert werden, wird auch die Funktion mehr als nur einmal in den Objektdateien auftauchen – ein Problem, das durch den Linker gelöst werden kann, wenn dieser dazu in der Lage ist. Er müsste nun die überzähligen Kopien der Funktion in den Objektdateien verwerfen. In vielen modernen Entwicklungsumgebungen findet das nicht statt. Oft ist es auch kein Problem, da man Speicher genug hat und ein paar zusätzliche Funktionen in der EXE-Datei keinen negativen Effekt haben. Wenn man allerdings wirklich auf Performanz und Codegröße[29] achten muss, sollte man das Verhalten des eingesetzten Compiler-Linker-Gespanns bezüglich der Inlinefunktionen genauer betrachten.

[28] Das Caching ist bei modernen Prozessoren eine Technik, die die Geschwindigkeit der Ausführung steigert. Ist der Code groß, ist das Caching aufwändiger.

[29] Die Größe eines Compilats nennt man auch „footprint“.

2.2.19 Makros

Die im Abschnitt 2.2.2 auf Seite 14 beschriebene Präprozessordirektive birgt noch einige Fähigkeiten mehr, als nur die Möglichkeit symbolische Konstanten zu definieren. So können mit der `#define`-Direktive Makros definiert werden, die Parameter entgegennehmen und damit Textersetzungen durchführen können. Ein einfaches Beispiel ist das folgende:

```
#define QUADRAT( x ) ( (x) * (x) )
```

Ein Beispiel für die Anwendung:

Listing 2.43. Definition und Anwendung eines Makros

```
#define QUADRAT( x )  ( (x) * (x) )

int main()
{
  int i = 7;
  int j = QUADRAT( i );
  // ...
  return 0;
}
```

Die Anwendung eines solchen Makros sieht aus wie der Aufruf einer Funktion. Man spricht deshalb auch von einem „Funktionsmakro“ oder einer „unechten Funktion“.

Für normalen Programmtext mag die Klammerung übertrieben erscheinen, in Makros ist sie jedoch wichtig. Der Präprozessor ist eine Compilerphase, die keine Typen kennt. Daher kann das `x` in dem Beispiel auch nicht in irgendeiner Form typisiert sein. Es ist einfach ein Stellvertreter für einen beliebigen Text, der durch den Präprozessor verarbeitet wird. Ob der Text syntaktisch an die Stelle passt, ist eine unabhängige Frage und wird auch nicht durch den Präprozessor überprüft. Die Textersetzung macht aus der Zeile:

```
j = QUADRAT( i + 2 );
```

die folgende:

```
j = ((i+2)*(i+2));
```

An dieser Stelle ist auch gut nachvollziehbar, warum die intensive Klammerung der Elemente des Makros wichtig ist. Wäre das Makro so definiert:

```
#define QUADRAT( x ) x*x
```

Dann sähe das Resultat von `j = QUADRAT( i + 2);` so aus:

```
j = i+2*i+2;
```

Das ist definitiv nicht, was man von einer Quasifunktion erwarten sollte. Denn sie reagiert ganz anders als eine Funktion. Die Textersetzung eines Makros birgt die Gefahr zu Nebeneffekten. Mit der korrekten Klammerung vermeidet man diese unliebsamen Nebeneffekte. Bei der Definition von Makros muss man sich allerdings bewusst sein, dass Nebeneffekte auch trotz der richtigen Klammerung noch auftreten können. Als Beispiel lässt sich wieder unser Makro `QUADRAT()` heranziehen:

```
j = QUADRAT( i++ );
```

Wäre `QUADRAT()` eine echte Funktion, würde der Wert aus `i` zur Berechnung des Quadratwertes herangezogen und danach `i` selbst um 1 erhöht werden. Das Makro bildet den Ausdruck `(i++)*(i++)`. Sowohl `i` als auch `j` hätten einen ganz anderen, wahrscheinlich erwartungswidrigen Wert.

Ein Makro darf beliebig viele Parameter haben, die wie in der Parameterliste einer Funktion mit Kommas getrennt werden. Es kann auch über mehrere Zeilen gehen. Dazu bringt man den umgekehrten Schrägstrich `\` – back slash – am Ende der Zeile an. Ein solcher back slash wird durch den Präprozessor immer als Fortsetzung der Programmzeile in der nächsten Textzeile interpretiert. Dabei muss dieses Zeichen an letzter Stelle in der Zeile stehen. Es darf auch kein Leerzeichen, kein Tabulator und kein Kommentar mehr folgen.

```
// Makro in zwei Zeilen
#define  QUADRAT( x )  \
         ( (x) * (x) )
```

2.2.20 Dynamische Speicherallokation

Bis jetzt wurden nur lokal und global deklarierte Daten besprochen. Diese Daten in Form von Variablen und Arrays haben alle eine Sache gemeinsam: Ihr Umfang, also ihre Größe, ist festgelegt. Wenn man an eine beliebige Anwendung, wie zum Beispiel einen Texteditor, eine Office-Applikation oder auch einen Internetbrowser denkt, dann kann man sich leicht vorstellen, dass diese Applikationen zur Laufzeit unterschiedliche Mengen an Daten verwalten müssen. Zur Laufzeit wird erst klar, wie viel Platz ein Texteditor für den aktuell geladenen Text benötigt. Würde man diesen Platz in Form eines Großen Arrays auf dem globalen Datenraum einer Applikation deklarieren, dann wäre damit die maximale Größe der zu bearbeitenden Texte festgelegt, unabhängig davon, wieviel Speicher das System tatsächlich besitzt. Außerdem würde auch dann viel Speicher durch die Anwendung in Anspruch genommen werden, wenn nur kleine Texte im Editor geladen wären. Da sich also die Größe des

benötigten Speichers zur Laufzeit entscheidet, muss es ein Verfahren geben, das Speicher dynamisch vom System anfordert und auch wieder freigibt.

Genau genommen handelt es sich um zwei unterschiedliche Vefahren, die in C++ Anwendung finden. Da ist zum einen das Funktionspaar `malloc()` und `free()` aus der Standardbibliothek der Sprache C. Da die C-Bibliothek in C++ zur Verfügung steht, können diese Funktionen und ihre Verwandten `calloc()` und `realloc()` weiter verwendet werden. Diese Funktionen gehören wie schon erwähnt zur Bibliothek und nicht zur eigentlichen Sprache. C++ hat gegenüber C eine Sprecherweiterung erfahren, die die dynamische Speicherallokation abdeckt. Das sind die beiden Operatoren `new` und `delete`. Die beiden Verfahren, das C-Verfahren und das C++-Verfahren, dürfen zwar parallel in einer Anwendung verwendet werden, mischen darf man sie allerdings nicht. Wenn man Speicher mit einem Verfahren anfordert, muss der Speicher auch mit dem entsprechenden Gegenstück des gleichen Verfahrens freigegeben werden. Aber nun zu einigen praktischen Beispielen:

Das C - Verfahren

Die beiden Funktionen `malloc()` und `free()` sind Bestandteile der Standard-C-Bibliothek. Man bindet sie in ein C-Programm ein, indem man die Headerdatei `stdlib.h` inkludiert. In standardkonformen C++-Programmen bindet man `cstdlib` ein.

Listing 2.44. Dynamische Allokation von Speicher in C

```
long *p;
p = (long *)malloc( 1000 * sizeof(long) );
...

...
free(p);
```

In diesem Beispiel wird ein Speicherblock alloziert, der die Größe von 1000 `long`-Variablen besitzt. Eigentlich wird an die Speicherverwaltung nur die Anforderung nach 4000 Bytes gesendet. Wie in der Anwendung die 4000 Bytes interpretiert werden, ist der Speicherverwaltung egal. Die Interpretation als eine Aneinanderreihung von `long`-Variablen geschieht durch die Typenkonvertierung `(long *)`, mit der der Rückgabewert der Funktion `malloc()` versehen wird. Die Funktion liefert per Deklaration den Typ `void *` zurück, was einem untypisierten Zeiger entspricht. Der Zeiger auf den Speicher ist deshalb untypisiert, damit jeder beliebige Datentyp in dynamisch alloziertem Speicher abgebildet werden kann. Durch die Typisierung legt man eigentlich nur eine Schablone über den Speicher, der einfach nur aus aneinandergereihten Bytes besteht. Die Funktion `malloc()` nimmt die Anzahl der geforderten Bytes entgegen. Die Formulierung `1000 * sizeof(long)` ist eigentlich nur für

den Entwickler wichtig, der seinen Code lesbar halten möchte[30]. Der Compiler macht einfach nur `4000` daraus.

Natürlich gibt es auch den Fall, dass die Speicherverwaltung nicht mehr genügend Speicher bereitstellen kann, um die Anforderung zu erfüllen. In diesem Fall gibt die Funktion `malloc()` den Zeigerwert `0` zurück, der immer einen ungültigen Zeiger anzeigt. Das Beispiel müsste also korrekterweise auf diesen Nullzeiger überprüfen:

Listing 2.45. Sicherung der dynamischen Speicherallokation

```
long *p;
p = (long *)malloc( 1000 * sizeof(long) );
if( p != 0 )
{
  ...
  free(p);
}
```

Der Nullzeiger ist in der Standardbibliothek als `NULL` definiert, sodass man auch die Überprüfung so schreiben kann: `if( p != NULL )`. Natürlich darf nur Speicher zugegriffen oder zurückgegeben werden, der auch wirklich alloziert wurde. Auf einen Nullzeiger kann die Funktion `free()` angewendet werden. Sie muss ihn ignorieren und ohne Reaktion zurückkehren.

Listing 2.46. Zugriff auf den allozierten Speicher

```
long *p;
p = (long *)malloc( 3 * sizeof(long) );
if( p != 0 )
{
  p[0] = 42;      /* schreibender Zugriff */
  p[1] = p[0]/7; /* Lese- und Schreibzugriff */
  p[2] = p[1]/2; /* Lese- und Schreibzugriff */

  free(p);
}
```

Natürlich wird bei der Nutzung der dynamischen Speicherallokation in realer Software der allozierende Teil und der freigebende Teil an ganz unterschiedlichen Stellen stehen. Die Funktionen `malloc()` und `free()` werden räumlich nicht so eng im Code zu finden sein wie in den aufgeführten Beispielen. Es ist gerade die Kunst in der Nutzung dieser Technik, die Software gewissermaßen „wasserdicht" zu gestalten. Es muss jeder allozierte Speicher wieder freigegeben werden. Andernfalls spricht man von Speicherlecks. Die Software

[30] Es ist dringend ratsam, den Code in solcher Weise lesbar zu halten. Wenn man Code warten möchte, hat man mit der genannten Formulierung einen Hinweis darauf, dass es sich um 1000 `long`-Speicherstellen handeln soll.

alloziert Speicher, den sie nicht mehr zurückgibt. Der Programmierer war vielleicht etwas nachlässig und hat Zeiger auf dynamischen Speicher einfach verloren, ohne die `free()`-Funktion aufzurufen.

Das C++-Verfahren mit `new` und `delete`

In C++ gibt es die Operatoren `new` und `delete`, die in gleicher Weise als Paar angewendet werden wie `malloc()` und `free()`. Dabei gibt es einige Unterschiede zu den Funktionen der C-Bibliothek. Das nächste – ANSI/ISO-konforme – Beispiel zeigt, dass schon die Fehlerbehandlung ganz anders funktioniert.

Listing 2.47. Dynamische Speicherallokation in C++ mit `new` und `delete`

```
long *p = NULL;
try
{
  p = new long[1000];
}
catch( std::bad_alloc &e )
{
}

...
delete [] p;
```

Außerdem muss der Zeiger, der durch den `new`-Operator zurückgegeben wird, nicht mehr extra konvertiert werden. Da der Operator zum Sprachbestand von C++ gehört, kann der Compiler den Datentyp überprüfen, für den die Allokation durchgeführt wird, und an den Zeiger weitergeben. Auch der Operator `delete` arbeitet mit dem Typ des übergebenen Zeigers. Im Beispiel wurde aber gleich ein Sonderfall gezeigt, der der Allokation eines Arrays. Dabei benutzen die Operatoren `new` und `delete` die eckigen Klammern `[]`. Vor allem beim Aufruf von `delete` dürfen sie im Fall des Arrays nicht vergessen werden, da sonst nicht alle Elemente des Arrays freigegeben werden. In späteren Abschnitten wird gezeigt werden, welche Unterschiede das Gespann `new` und `delete` in C++ gegenüber `malloc()` und `free()` eigentlich hat. Es kann aber schon etwas vorgegriffen werden mit dem Hinweis, dass `new` den Speicher nicht nur alloziert, sondern ihn auch initialisiert. Ebenso führt der Operator `delete` Aufräumarbeiten durch, bevor der reine Speicher an die Speicherverwaltung zurückgegeben wird. Auf diese Aspekte der Objektorientierten Programmierung wird in den Abschnitten 2.2.21, 2.2.21 sowie 2.2.23 näher eingegangen werden. Ein kleines Beispiel soll hier aber trotzdem schon gezeigt werden:

Listing 2.48. Dynamische Allokation eines einzelnen Objekts

```
long *p = NULL;
try
{
  p = new long(42);
  std::cout << *p << std::endl;
}
catch( std::bad_alloc &e )
{
}

...
delete p;
```

In diesem Beispiel werden nicht 42 Speicherstellen alloziert, sondern eine Speicherstelle des Typs `long`, die mit dem Wert 42 initialisiert wird. Man spricht auch von einem Objekt des Typs `long`. Der Operator `delete` benötigt in diesem Beispiel keine eckigen Klammern `[]`, da es sich nur um ein zu löschendes Objekt und nicht um ein Array handelt. Er löscht exakt die vier Bytes, die der Operator `new` vorher allozierte. Um ganz genau zu sein: es gibt die Versionen `new` und `new[]`, die zwei unterschiedliche Operatoren sind. Dazu passend gibt es die Operatoren `delete` und `delete[]`. Die Operatoren `new` und `delete` sowie `new[]` und `delete[]` gehören paarweise zusammen und dürfen nicht gemischt werden. Genauso wenig, wie `malloc()` und `free()` mit einem der C++-Operatoren zur dynamischen Speicherverwaltung gemischt werden dürfen.

Fragmentierung des Speichers

Dynamisches Allozieren und Deallozieren von Speicher bringt ein Problem mit sich, das allgemein als Speicherfragmentierung bekannt ist. In bestimmten Fällen kann die Speicherfragmentierung zu Laufzeitfehlern der Software führen. Betrachten wir also zunächst das Phänomen in einer etwas vereinfachten Darstellung, um später zu den Konsequenzen zu kommen: Der Speicher ist für die Software, die auf ihn zugreift, linear organisiert. Das heißt, dass die Speicherzellen einfach eine fortlaufende Adressierung haben. (Die Technik der virtuellen Speicherverwaltung moderner Betriebssysteme soll erst später mit in Betracht gezogen werden.) Wird ein Speicherblock alloziert, geschieht das vorzugsweise am Anfang des fortlaufenden Gesamtspeichers. Der nächste zu allozierende Block wird dann direkt an den ersten angeschlossen, sodass sich die Menge des freien Speichers von den niedrigen zu den hohen Adressen reduziert. Wird nun der erste Block von der Software wieder freigegeben, nimmt die Menge des freien Speichers wieder zu. Dieser ist jedoch nicht mehr an einem Stück zusammenhängend. Er ist fragmentiert.

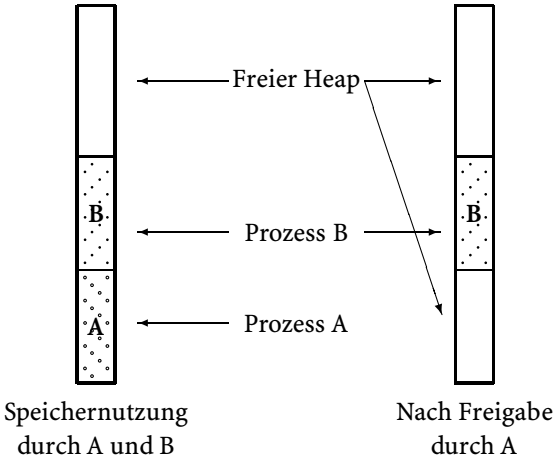


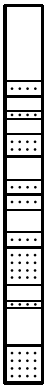
Abbildung 2.1. Allokation und Deallokation bei physikalischer Speicherverwaltung

Die Fragmentierung wird genau dann zum Problem, wenn ein Speicherblock alloziert werden soll, der zwar theoretisch noch in den freien Speicher passt, praktisch aber größer ist als der größte zusammenhängende freie Speicherbereich. Bei einer ungünstigen Abfolge von Allokierungen und Deallokierungen kann der verbliebene freie Speicher in lauter kleine Stücke zerteilt werden, was die Wahrscheinlichkeit eines Laufzeitfehlers erheblich steigert.

In der beschriebenen einfachen – und problematischen – Form tritt die Speicherfragmentierung vor allem bei Systemen mit physikalischer Speicherverwaltung auf. Das heißt, wenn Systeme die Adressierung des Speichers einfach an die physikalische Reihenfolge anlehnen, die der Speicher auf den Speicherchips hat. Auf praktisch allen Mikrocontrollern ohne Betriebssystem ist das der Fall. Ältere Betriebssysteme wie DOS, CM/P und einige Derivate adressieren auch physikalisch. Alle modernen UNIX-Varianten und Derivate, Windows NT/2000/XT und Linux haben eine virtuelle Speicherverwaltung, die neben einer sauberen Prozesstrennung auch die Speicherfragmentierung reduzieren hilft.

Wie wird also die Speicherfragmentierung durch eine virtuelle Speicherverwaltung reduziert? Und welche Fragmentierung bleibt auch bei virtuellen Speicherverwaltungen bestehen? Das sind eigentlich Fragen nach der Technik von Betriebssystemen, die eigentlich keine Auswirkungen auf sprachliche Strukturen in C++ haben. Dennoch handelt es sich hierbei um Hintergrundwissen, das für einen erfolgreichen Einsatz von C++ in Softwareprojekten unerlässlich ist. Gehen wir also kurz auf dieses Betriebssystemwissen ein.

Ein System mit virtueller Speicherverwaltung weist einem Prozess Speicher zu, den es auf eine von den physikalischen Adressen unabhängige Weise



Speicher mit vielen belegten und freien Blöcken nach längerer Nutzung.

Abbildung 2.2. Fragmentierter Speicher

adressiert. Wenn also ein Programm einen Speicherblock alloziert, bekommt es einen Adresswert, der nicht die physikalische Position des Speichers auf den Speicherchips beschreibt. Dieser Adresswert wurde durch das Betriebssystem vergeben und nur der Prozess, in dem das Programm läuft, kann auf diese Adresse zugreifen. Bei einem Zugriff muss diese virtuelle Adresse in die physikalische übersetzt werden. Auch das übernimmt das Betriebssystem oder ein Hardwarebaustein auf der CPU mit dem Namen Memory Management Unit – MMU. Das Programm selbst hat keine Kenntnis über die zugrunde liegende physikalische Adresse. Das Betriebssystem oder die MMU verwaltet die Tabellen, die zur Übersetzung der virtuellen in die physikalischen Adressen notwendig sind[31]. An diesen Tabellen ist auch vermerkt, welche Prozesse auf welche virtuellen – und damit auch physikalischen – Adressen Zugriff haben. Versucht ein Prozess auf eine Adresse zuzugreifen, die ihm nicht explizit vom Betriebssystem zugewiesen wurde, so führt das zum Abbruch des Prozesses. Das Betriebssystem wacht also über das korrekte Verhalten der ablaufenden Prozesse. Aber was hat das nun mit der Reduktion der Speicherfragmentierung zu tun? Wenn ein Programm einen Speicherblock anfordert, bekommt es eine Adresse auf die erste Stelle in einem zusammenhängenden Bereich. Wäre es eine physikalische Adresse, müsste der gesamte Bereich auch physikalisch zusammenhängen. Ein virtuell durchgängig adressierter Bereich muss jedoch nicht physikalisch zusammenhängen. Er kann aus mehreren getrennten physikalischen Blöcken bestehen. Wichtig ist nur, dass die Übersetzung der virtuellen in die richtige physikalische Adresse im jeweils richtigen physikalischen Block erfolgt. Die virtuelle Speicherverwaltung eines Betriebssystems ist also in der Lage, physikalisch getrennte Blöcke in einem physikalisch frag-

[31] Wenn eine MMU vorhanden ist, nennt sich deren Speicherbereich, in dem sich die Umsetzungstabellen für die Adressen befinden, der Translation Lookaside Buffer.

mentierten Speicher zu zusammenhängenden virtuellen Blöcken zu formen und damit der Fragmentierung entgegenzuwirken.

Der Haken an der Sache, weshalb eine Fragmentierung dadurch nicht vollständig verhindert werden kann, ist folgender. Würde die Speicherverwaltung des Betriebssystems jede beliebige physikalische Blockgröße mit virtuellen Adressen ausstatten, könnte der Verwaltungsaufwand selbst mehr Speicher für die Verwaltung besagter Tabellen benötigen, als insgesamt Speicher für Anwendungen verwaltet wird[32]. Die verwalteten Blöcke haben also eine Größe, die in einem sinnvollen Verhältnis zum Verwaltungsaufwand steht und die in einer effektiven Weise durch die Hardwaren (CPU, Speicherbausteine und Bussystem) transportiert werden kann. Die Blöcke nennt man Speicherseiten und deren Größe die Seitengröße. Die Seitengröße beträgt bei den meisten Systemen 4096 oder 8192 Bytes[33]. Es werden durch die Speicherverwaltung des Betriebssystems immer vollständige Speicherseiten an einen Prozess übergeben. Dieser fügt sie dann zu größeren Bereichen zusammen oder er zerteilt Seiten in kleinere Bereiche. Genau dort finden wir auch die Fragmentierung wieder, denn kleinere Bereiche, die eine Seite zwar belegen, diese aber nicht vollständig ausfüllen, verschwenden den Rest der Seite. Es sind Fragmente unterhalb der Seitengröße, die weiterhin existieren. Was bei einem System mit virtueller Speicherverwaltung aber nicht mehr passiert, ist beispielsweise die Zerschneidung des gesamten Speichers in zwei oder mehrere ähnlich große Bereiche. Die verbleibende Fragmentierung des Speichers bei virtueller Speicherverwaltung betrifft also Blockgrößen unterhalb der Seitengröße. Auch dafür gibt es einige Lösungen, die meistens auf vorallozierte Speicherpools basieren, die bestimmte Blockgrößen aufnehmen können. Es gibt Betriebssysteme, die in ihrer Standardimplementierung der Prozessspeicherverwaltung solche Poolingverfahren verwirklichen[34]. Diese Poolingverfahren genauer zu beschreiben, sprengt zumindest den Rahmen dieses Buchs. Es kann aber zusammengefasst werden, dass der Programmierer nur noch selten mit der Problematik der Speicherfragmentierung in Berührung kommt, wenn er mit einem Betriebssystem arbeitet, das eine virtuelle Speicherverwaltung besitzt. Bei der Entwicklung von Embedded-Systemen ist die Fragmentierung ein Problem, das dem Entwickler bekannt sein sollte, wenn er Entscheidungen bezüglich der Speichernutzung fällt.

[32] Man stelle sich den Extremfall vor, in dem jeder Block der Größe 1 in einer Tabelle aufgeführt werden würde mit einer entsprechenden virtuellen Adresse. In diesem Fall würde die Tabelle ein Vielfaches des verwalteten Speichers benötigen.

[33] Also 4KB oder 8KB. Grundsätzlich können es auch Vielfache davon sein.

[34] QNX 6.x ist ein solches Betriebssystem, das die verbliebene Fragmentierung auf Seitenebene mit Hilfe eines an den Blockgrößen in Zweierpotenzen ausgerichteten Poolingverfahrens reduziert.

2.2.21 Strukturen und Klassen

Neben den Standarddatentypen gibt es auch die zusammengesetzten Datentypen, die das Objektorientierte Programmieren eigentlich ausmachen. Das Verständnis dieser Programmstrukturen ist also wesentlich. Deshalb wird dieses Unterkapitel so aufgebaut, dass die Entwicklung der Idee der zusammengesetzten Datentypen nachvollzogen werden kann. Zunächst geht es zur Struktur in C, um danach zur Klasse in C++ überzuleiten.

Die Struktur in C

In der C-Programmierung wurde das prozedurale Paradigma zur Anwendung gebracht. Die Sprache C unterstützt dieses Prinzip der Strukturierung von Software in einer Notation, die entfernt an die Notation der Funktionsmathematik, der Analysis, erinnert. Diese Idee der Programmstrukturierung auf Basis des Begriffs der Funktion[35] unterstützen auch andere Sprachen, die daher der Familie der prozeduralen Sprachen zugerechnet werden. Neben C sind das unter anderen auch Pascal und Fortran. Ein Merkmal dieses prozeduralen Prinzips ist die strikte Trennung zwischen Daten und Funktionen. In einer frühen Phase der Informationstechnologie wurde gerade damit geworben, dass die Anwendung dieser Trennung ein großer Vorteil sei, denn er ermögliche ein unabhängiges Variieren der statischen und der dynamischen Anteile eines Programms. Außerdem mache diese Trennung die Gesamtstruktur der Software besser verstehbar. Zur Modellierung des dynamischen Teils wurde die Funktion (bzw. die Prozedur) eingeführt. Zur Modellierung des statischen Teils wurde die Struktur eingeführt[36].

Die Struktur fasst Daten zusammen, die aus irgendeinem fachlichen Grund zusammengehören. So gehören beispielsweise in einem zweidimensionalen Koordinatensystem die beiden Koordinaten eines Punktes X und Y zusammen und man kann daraus eine Struktur bilden. Die Zusammenfassung erleichtert die Arbeit mit den Daten, da diese immer in einer aufeinander bezogenen Weise verwendet werden.

Listing 2.49. Eine einfache Struktur

```
struct Punkt
{
  int x
  int y;
};
```

[35] bzw. der Prozedur.

[36] Dieser Gedanke vereinfacht etwas, denn es gibt in C – und nicht nur dort – noch andere Möglichkeiten der Definition eigener Datentypen. So gibt es die Unions und die Enumerationstypen.

Strukturen können verschachtelt werden. So kann beispielsweise ein Kreis in einem solchen zweidimensionalen Koordinatensystem unter Zuhilfenahme des Punktes definiert werden.

Listing 2.50. Eine verschachtelte Struktur

```
struct Kreis
{
  struct Punkt m;
  int r;
};
```

Zum Zugriff auf die Elemente einer Instanz der Struktur wurde der Punktoperator bestimmt.

Listing 2.51. Zugriffe auf Strukturelemente

```
struct Kreis k1
k1.r = 5;
k1.m.x = 0;
k1.m.y = 0;
```

Es lassen sich auch Zeiger auf Strukturen bilden. Dazu verwendet man in der Deklaration den Stern, wie bei den Standarddatentypen auch.

Listing 2.52. Anwendung des Pfeiloperators

```
struct Punkt p1;
struct Punkt *pp;
pp = &p1;
pp->x = 0;
pp->y = 0;
```

Um über einen solchen Zeiger zuzugreifen, muss er natürlich auf eine gültige Speicherstelle mit dem Inhalt eines Punktes verweisen. Greift man dann auf Elemente der Struktur zu, kommt dafür der Pfeiloperator `->` zum Einsatz.

Zeiger auf zusammengesetzte Datenstrukturen werden besonders häufig für die Parameterübergabe in Funktionen verwendet. Würde die Struktur selbst übergeben werden, würde das unverhältnismäßig viel Laufzeitaufwand bedeuten, da alle Daten der Struktur auf den Stack kopiert werden müssten, um der Funktion einen Zugriff darauf zu ermöglichen. Darüber hinaus könnte die Funktion die Daten zwar lesen, aber nicht beschreiben, denn sie hat eine Kopie der Daten bekommen. Über einen Zeiger kann eine Funktion auch schreibend auf eine Datenstruktur zugreifen. Beim Funktionsaufruf selbst wird nur ein Zeiger[37] übergeben.

[37] Also 2, 4, 6 oder 8 Bytes. Je nach Systemplattform.

Listing 2.53. Übergabe eines Strukturzeigers an eine Funktion

```
void KreisInitialisierung( struct Kreis *k )
{
  k->m.x = 0;
  k->m.y = 0;
  k->r = 0;
}
```

Häufig findet man auch Zeiger in Parameterlisten, die durch das Schlüsselwort `const` qualifiziert sind. Solche Funktionen erklären damit, dass sie nur lesend und keinesfalls schreibend auf die per Zeiger übergebenen Daten zugreifen. Man spricht dann von einer „Const-Maskierung" des Parameters. Der Compiler verbietet in diesem Fall auch den schreibenden Zugriff auf den Parameter innerhalb der Funktion.

Listing 2.54. Const-Maskierung eines Strukturzeigers

```
double KreisUmfang( const struct Kreis *k )
{
  return 2.0 * k->r * 3.141;
}
```

Solche Const-Maskierungen sollten genutzt werden. Im Abschnitt 2.2.34 auf Seite 141 wird dieses Thema gründlich behandelt.

Das Klassenkonzept

Die Klasse ist technisch gesehen einfach eine Struktur, die in einer bestimmten Hinsicht erweitert wurde. So können in eine Klasse neben den Datenelementen auch Funktionen aufgenommen werden. Diese Funktionen innerhalb von Klassen nennt man Methoden, die Datenelemente Attribute. In den folgenden Abschnitten soll näher auf die syntaktischen Regeln im Zusammenhang mit der Klasse in C++ eingegangen werden. Zunächst soll die Klasse jedoch aus einer etwas theoretischen Sicht beleuchtet werden.

Strukturen wurden schon in C zur Strukturierung von Daten verwendet. Haben Daten eine gewisse Zusammengehörigkeit, kann man das durch die Bildung einer Struktur unterstreichen, was einen Quelltext idealerweise besser lesbar macht. So kann zum Beispiel das Datenelement x mit dem Datenelement y zu einer Datenstruktur namens „Punkt" kombiniert werden, wenn man x und y in Sinne von Punktkoordinaten in einer Ebene verwenden möchte. Der Quelltext wird damit quasi auf eine höhere Abstraktionsebene gehoben, wenn in ihm von Punkten und nicht von separierten x und y die Rede ist. Das Klassenkonzept in C++ geht diesen Weg konsequent weiter. Die Klasse „klassifiziert" in dem Sinne, wie es die Datenstruktur vor ihr schon getan hat. Ihre Möglichkeiten zur Klassifikation sind nur ungleich größer als

die der Struktur in C[38]. Die Verwendung von Klassen in einem Programm zielt also hauptsächlich darauf, ein höheres Abstraktionsniveau in der Ausformulierung des Quelltextes zu erlangen. Ihre Ausdrucksmächtigkeit gegenüber den Elementen der Prozeduralen Programmierung gewinnt die Klasse durch die Möglichkeit, Methoden – also Funktionen – aufzunehmen. Damit wird die Funktion als das wichtigste strukturelle Element prozeduraler Programmentwürfe entthront. Tendenziell werden in der prozeduralen Programmierung die Daten den Funktionen in Form von Parametern „zugeordnet". In der Objektorientierten Programmierung werden die Funktionen den Daten in Form der Klasse zugeordnet. Das schafft Möglichkeiten, die weit über die der herkömmlichen prozeduralen Programmierung hinausgehen. Diese Umkehr der Betrachtungsweise rückt die Funktion aus dem Brennpunkt des Interesses und gibt den Blick auf die Daten frei, über die nun intensiver nachgedacht werden kann. Man kann die Daten nun besser klassifizieren, d. h. man schafft sich die Klassen, die die Daten am besten repräsentieren, und ordnet ihnen die nötigen Methoden zu. Klassen sollten dabei für gut umrissene Begriffe in der Beschreibung des Programms stehen. Dazu aber in späteren Kapiteln dieses Buchs mehr. Hier soll zunächst auf die Syntax eingegangen werden.

Ausgehend von der Struktur in C gibt es für Klassen vor allem zusätzliche Regeln. Die alten Regeln aus C bleiben auch für die Klassen mit zwei wichtigen Ausnahmen bestehen:

1. In C++ braucht man für die Verwendung eines definierten Strukturtyps das Schlüsselwort `struct` nicht zu wiederholen. Das gilt auch für die Klasse. Das Schlüsselwort `class` wird bei der Definition einer Klasse gebraucht, nicht jedoch beim Anlegen oder bei der Deklaration von Daten.
2. Elemente in einer Klasse unterliegen sogenannten Sichtbarkeitsregeln. Diese Regeln definieren, wann und wo man auf welche Elemente zugreifen kann. In C kann man auf Strukturelemente immer zugreifen, wenn eine Struktur fassbar ist.

Zur Wiederholung noch einmal das Beispiel mit dem Punkt:

Listing 2.55. Noch einmal der Punkt

```
struct Punkt
{
  int x
  int y;
};
```

In C++ kann eine Variable des Punkts folgendermaßen instanziiert werden:

```
Punkt p;
```

In C sähe die Deklaration so aus:

[38] Konsequenterweise ist die Struktur in C++ eine Klasse mit unwesentlich veränderter Syntax.

```
struct Punkt p;
```

Es muss also in C++ nicht wie in C klargestellt werden, dass es sich bei dem Bezeichner `Punkt` um eine Struktur handelt. C++ akzeptiert die Definition eines Bezeichners als Typ, ohne weitere zusätzliche Anforderungen für deren Anwendung festzulegen. Ein solcher Punkt lässt sich nun auch als Klasse definieren:

Listing 2.56. Eine einfache Klasse

```
class Punkt
{
  int x
  int y;
};
```

Würde man nun eine Variable dieses Typs anlegen und versuchen auf ihre Datenelemente zuzugreifen, wäre dieser Versuch nicht von Erfolg gekrönt, denn es gibt ja noch den zweiten Unterschied zur Struktur in C. Dieser Unterschied betrifft die Sichtbarkeitsregeln für die Elemente einer Klasse. Wenn nichts anderes definiert ist, sind Elemente standardmäßig privat. Das heißt, dass sie nicht von außerhalb der Klasse zugegriffen werden können.

Listing 2.57. Zugriffsversuch auf die Elemente der Klasse

```
Punkt p;
p.x = 0; // Compilefehler!
p.y = 0; // Compilefehler!
```

Wird die Klasse ein klein wenig anders definiert, können die Elemente wie Strukturelemente zugegriffen werden:

Listing 2.58. Öffnen der Sichtbarkeit der Attribute

```
class Punkt
{
public:
  int x
  int y;
};

Punkt p;
p.x = 0; // OK!
p.y = 0; // OK!
```

In C++ sind also alle Klassenelemente standardmäßig privat und müssen bei Bedarf einem Zugriff von außen geöffnet werden. Die Struktur in C++ unterscheidet sich nur in dem einzigen Merkmal, dass ihre Elemente standardmäßig öffentlich, also von außen zugreifbar sind. Diese öffentliche Sichtbarkeit der Struktur kann aber auch eingeschränkt werden.

Attribute

Attribute sind Datenelemente in Klassen. Man deklariert sie wie die Elemente in einer Struktur. Wenn keine explizite Sichtbarkeit angegeben ist, sind die Elemente einer Klasse privat. Ein Grundprinzip der Objektorientierten Programmierung lautet, dass Attribute in Objekten privat sein und Zugriff darauf über die Methoden des Objektes erfolgen sollten. Dieses Grundprinzip wird auch das Delegationsprinzip genannt. Das Objekt ist in der Verantwortung, die Konsistenz der attributären Werte zu gewährleisten. Deshalb delegiert man die Aufgaben auch an das Objekt, das die Attribute enthält.

Listing 2.59. Klassenattribute

```
class Kreis
{
public:
  int x, y;      // Attribute
  double radius; // Attribut
};
```

Wie man Methoden für eine Klasse definiert, um damit das Delegationsprinzip zu gewährleisten, wird in Abschnitt 2.2.21 ab Seite 73 besprochen.

Konstante Attribute

Konstante Attribute einer Klasse werden wie variable Attribute deklariert, allerdings mit dem Schlüsselwort `const`. Das Besondere an den konstanten Attributen ist allerdings, dass sie initialisiert werden müssen. Das geschieht in der Initialisierungsliste des Konstruktors (mehr dazu im Abschnitt 2.2.21 auf Seite 80.). Die konstanten Attribute vergrößern den Speicherplatz eines Objekts. Da nun eine Konstante unabhängig von der Anzahl der instanzierten Objekte immer den gleichen Wert besitzt, sollte man sich fragen, ob eine solches konstantes Attribut nicht statisch definiert werden sollte. Damit wird es nur einmal im Speicher abgelegt und kann durch alle Objekte des entsprechenden Typs zugegriffen werden. Die Initialisierung der statischen Konstanten erfolgt bei deren Definition[39].

Natürlich ist die Definition eines konstanten Objektattributs in einer Klasse nicht ganz gleichbedeutend mit der Definition eines statischen konstanten Attributs. Bei unterschiedlicher Vorbelegung in der Initialisierungsliste der Objekte können konstante Objektattribute für jedes Objekt einen anderen Wert besitzen.

Neben den beiden genannten Techniken gibt es noch die Möglichkeit einen konstanten Wert in einer Klasse zu definieren. Es wird dazu ein anonymer `enum`-Typ definiert, der den konstanten Wert mit Zuweisung enthält.

[39] Siehe Abschnitt 2.2.28 auf Seite 112.

Im Allgemeinen ist diese Variante der vorzuziehen, die ein statisches Klassenelement benötigt. Es verbraucht keinen Speicher und verlagert das Problem des konstanten Wertes auf den Compiler. Wenn allerdings eine Adresse der Konstanten gebraucht wird, ist eventuell die Variante mit dem statischen Element die bessere. Die Definition einer symbolischen Konstante über ein `enum`-Konstrukt ist kein Attribut einer Klasse.

Listing 2.60. Die drei Möglichkeiten der Konstantendefinition in Klassen

```
class X
{
public:

  X() : KONSTANTE1(42) {}

  const int KONSTANTE1;

  static const int KONSTANTE2;

  enum{ KONSTANTE3 = 42 };
};

const int X::KONSTANTE2 = 42;
```

Methoden

Methoden sind Elementfunktionen von Klassen. Man hat sich in der Objektorientierten Programmierung einfach auf diesen Namen verständigt. Methoden sind es auch, die die Schnittstelle einer Klasse und damit all ihrer Objekte festlegen. Objekte interagieren mit Methoden. Und Methoden ändern die Attribute der Objekte.

Sie werden deklariert, indem man ihre Prototypen einfach in die Klasse schreibt, zu der sie gehören sollen.

Listing 2.61. Methoden

```
class Kreis
{
public:
  double Umfang();
  double Flaeche();

  int x,y;
  double r;
};
```

Auch ist darauf zu achten, dass Methoden in der richtigen Sichtbarkeit stehen, d. h., dass sie in dem Block unterhalb des gewünschten Sichtbarkeitsbezeichners deklariert sind. Zur Sichtbarkeit mehr im Abschnitt 2.2.21. Definiert werden die Methoden üblicherweise außerhalb ihrer Klassen. Die Zugehörigkeit zu ihren Klassen muss aber bei der Definition bekräftigt werden. Dazu wird der sogenannte „Scope-Operator“ `::` verwendet.

Listing 2.62. Implementierung von Methoden

```
double Kreis::Umfang()
{
  return 2.0 * r * 3.141;
}

double Kreis::Flaeche()
{
  return 3.141 * r * r;
}
```

Innerhalb der Methoden kann direkt auf Klassenelemente zugegriffen werden. Syntaktisch befindet man sich innerhalb der Methodenrümpfe in den Namensräumen der entsprechenden Klassen. Deshalb sind auch alle Elemente der Klasse zugreifbar, auch die privaten.

Technisch steckt ein einfaches Verfahren dahinter, immer die richtigen Daten des entsprechenden Objekts zugreifen zu können. Denn die Methoden können jetzt an Instanzen von Klassen, also an Objekten aufgerufen werden.

Listing 2.63. Methodenzugriff

```
Kreis k1;
k1.x = 0;
k1.y = 0;
k1.r = 2;
double u = k1.Umfang();
double f = k1.Flaeche();
```

Wie kommen die Methoden also rein technisch gesehen an die Daten der Objekte? Schließlich könnte in dem Beispiel die Methode `Kreis::Umfang()` auch an einem Objekt `k2` aufgerufen werden, das wahrscheinlich ganz andere Werte enthalten würde als `k1`. Der Trick ist, dass die Methoden einen Parameter bekommen, den man im C++-Quelltext nicht sehen kann. Man spricht deshalb auch vom impliziten Parameter. Dieser implizite Parameter steht vor den anderen auf dem Stack. Im Assemblercode, den ein Compiler aus dem C++-Quelltext erzeugen kann, sieht man, dass die Methoden immer einen – den ersten – Parameter mehr haben, als in C++ deklariert. Dieser Parameter ist typisiert mit einem Zeiger auf den eigenen Klassentyp.

Der `this`-Zeiger

Für die Methoden `Kreis::Umfang()` und `Kreis::Flaeche()` ist der implizite Parameter also mit `Kreis*` typisiert. Dieser Zeiger kann auch angesprochen werden mit dem extra dafür vorgesehenen Schlüsselwort `this`. Der `this`-Zeiger hat also immer den Typ der Klasse, in deren Namensraum man sich befindet. Er zeigt auf das Objekt, für das eine Methode aufgerufen wird. Anders formuliert, er zeigt auf das „eigene" Objekt einer Methode.

Das Listing 2.62 kann man also folgendermaßen umschreiben:

Listing 2.64. Der `this`-Zeiger

```
double Kreis::Umfang()
{
  return 2.0 * this->r * 3.141;
}

double Kreis::Flaeche()
{
  return 3.141 * this->r * this->r;
}
```

Damit wird sichtbar gemacht, was der Compiler im Verborgenen ersetzt. Das Schlüsselwort `this` in C++ bezeichnet den implizit übergebenen Zeiger auf das Objekt. Über diesen Zeiger werden in einer Methode auch die Klassenattribute erreicht.

Das Delegationsprinzip

Methoden sind dazu da, die Funktionalität der Objekte einer Klasse zu definieren. Dabei greifen sie auf Attribute zu und verändern diese gegebenenfalls. Das Prinzip der ausschließlichen Verantwortlichkeit einer Klasse für ihre Attribute wird dadurch realisiert, dass man nur in Methoden der Klasse die Attribute verändert. Es sollte nicht von Methoden oder Funktionen außerhalb der Klasse geschehen. Man delegiert die verändernden Zugriffe immer an die Klasse. Daher auch der Name: Delegationsprinzip.

Die Einhaltung des Delegationsprinzips garantiert die weitgehende Entkopplung einer Klasse von anderen Klassen. Und genau diese Entkopplung sollte man in einem objektorientierten Entwurf anstreben, denn das vereinfacht ihn. Je entkoppelter die Einzelbestandteile eines Entwurfs sind, desto einfacher sind sie zu warten und zu erweitern. Wenn Klassen nicht nach dem Delegationsprinzip entworfen werden, können sie keinen konsistenten Objektzustand garantieren. In diesem Fall steigt auch die Fehleranfälligkeit des Codes.

Inline - Methoden

Methoden können wie globale Funktionen inline-expandiert werden. Dazu müssen sie nur innerhalb der Klasse definiert werden:

Listing 2.65. Inlining von Methoden

```
class X
{
public:
  X() { /* ... */ } // Inline Konstruktor
  ~X() { /* ... */ } // Inline Destruktor
  void f() { /* ... */ }  // Inline Methode
};
```

Eine weitere Möglichkeit ist die Verwendung des Schlüsselwortes `inline` bei der Definition der Methode. Auch für die Inline-Methoden gelten alle Überlegungen, die für globale Inline-Funktionen gelten. Es können grundsätzlich alle Methoden inline deklariert werden; also auch Konstruktoren und Destruktoren. Problematisch wird es bei der Inline-Deklaration von virtuellen Methoden. Virtuelle Methoden müssen über die V-Tabelle erreichbar sein. Es muss also ein Funktionspointer auf sie zeigen können, denn die V-Tabelle enthält Funktionspointer. Wenn ein Zeiger auf sie zeigt, kann die Funktion nicht expandiert worden sein, sondern hat einen regulären Platz im Speicher. Es ist aus diesem Grund nicht ratsam, virtuelle Methoden inline zu deklarieren. Man erhält den Effekt, dass Methoden mehrfach in verschiedenen Objektdateien vorkommen können, wenn die in Headerdateien definierten Methoden in mehreren Modulen inkludiert wurden. Man sollte also die Inline-Modifikation von virtuellen Methoden[40] vermeiden.

Kapselung und Sichtbarkeit

Der Kapselungsgedanke ist einer der zentralen Konzepte der Objektorientierung. Beim Zusammenfassen von Daten und Methoden in die Einheiten der Klassen wird zwischen Elementen unterschieden, die nach außen bekannt sein müssen, und solchen, die nur für die Klasse selbst notwendig sind. Diese systematische Grenzziehung zwischen Klassenelementen und Außenwelt nennt man Kapselung. Unter Kapselung kann man auch Systembildung verstehen. Ein System ist vor allem eine Grenze zwischen einer Systeminnenwelt und einer Außenwelt, wobei einige Elemente des Systems als Schnittstelle nach außen fungieren. Diese Elemente müssen nach außen „sichtbar“ sein. So wie die Knöpfe und Schalter eines CD-Players nach außen sichtbar und zugreifbar sind, im Gegensatz zu den nach außen nicht sichtbaren internen Elementen der Elektronik, die wesentlich für die Funktionalität des Players sind. Die syntaktischen Möglichkeiten in C++, Zugriffe auf Attribute und

[40] Siehe Abschnitt 2.2.30 auf Seite 121.

Methoden zu beschränken, unterstützt das Delegationsprinzip, das in den Abschnitten über Attribute (2.2.21) und über Methoden (2.2.21) schon angesprochen wurde.

Um die Sichtbarkeit von Elementen festzulegen, hat C++ drei Schlüsselwörter und einige syntaktische Regeln. Diese haben viel Ähnlichkeit mit den Regeln in anderen objektorientierten Sprachen, haben aber auch ihre spezifische Besonderheit.

Sichtbarkeit von Klassenelementen

In C++ gibt es drei Sichtbarkeiten von Klassenelementen, `public`, `protected` und `private`. Die Sichtbarkeit `public` definiert, dass jeder von außerhalb der Klasse auf das entsprechende Element zugreifen darf. Das Gegenteil dazu definiert `private`. Nur Methoden der Klasse, in der ein privates Element definiert ist, dürfen auf dieses zugreifen. Die Spezifikation `protected` steht im Zusammenhang mit der Vererbung und definiert, dass alle Methoden der Klasse, in der das Element definiert ist, auf das Element zugreifen dürfen, und darüber hinaus noch alle Methoden von Kindklassen, sofern keine private Vererbung durchgeführt wurde (siehe nachfolgenden Abschnitt).

In dem folgenden Beispiel wird die Verwendung der Sichtbarkeitsbezeichner gezeigt. Sie leiten jeweils einen Block der entsprechenden Sichtbarkeit ein, der nur durch einen anderen Sichtbarkeitsbezeichner unterbrochen werden kann, oder eben durch das Ende der Klasse. Ohne Sichtbarkeitsbezeichner ist der Klassenscope standardmäßig privat.

Listing 2.66. Die Sichtbarkeiten von Klassenelementen

```
class X
{
  int a; // a ist privat wie d
public:
  int b;
protected:
  int c;
private:
  int d;
};
```

Die Struktur in C++

Die Struktur in C++ ist eine spezielle Form der Klasse. Ihr Scope ist im Gegensatz zur Klasse standardmäßig öffentlich und nicht privat. Ansonsten sind Strukturen in C++ ganz und gar Klassen. Sie können vererbt werden, selber vererben und auch alle anderen syntaktischen und konzeptionellen Merkmale von Klassen sind vorhanden. In den meisten Quellcodes werden Strukturen allerdings wie die Strukturen aus C angewendet. Häufig stellen Strukturen

auch eine Schnittstelle zwischen C- und C++-Code dar. Wenn strukturierte Daten zwischen C- und C++-Modulen ausgetauscht werden sollen, kann man auf die Struktur zurückgreifen. Allerdings darf man keine sprachlichen Elemente aus C++ dabei verwenden.

Sichtbarkeit der Vererbung

Das Kapselungsprinzip stand zu Beginn des Einsatzes objektorientierter Methoden stark im Zentrum der Theoriediskussion. Das führte bei C++ als eine der alten OO-Sprachen zu einer sehr starken Gewichtung des Gedankens. Unter anderem ist die standardmäßig private Sichtbarkeit der Vererbung ein Produkt aus dieser Zeit. In moderneren OO-Sprachen wie z. B. Java wurde ganz auf eine Sichtbarkeitseinschränkung der Vererbung verzichtet, da diese aus heutiger OO-Sicht konzeptionell keine Bedeutung hat und sogar kontraproduktiv ist. Es mag nun sein, dass bestimmte Templatetechniken gerade dieses totgeglaubte Feature von C++ zu neuem Leben erwecken. Mit Objektorientierung hat das aber nichts zu tun und folgt im Allgemeinen ganz eigenen konzeptionellen Regeln. Um aber das Verhalten von C++ in diesem Punkt zu verdeutlichen, sollen einige Listings Klarheit schaffen:

Listing 2.67. Standardmäßige private Vererbung

```
class A
{
public:
  int x;
};

class B : A
{
};
```

Ohne Sichtbarkeitsbezeichner handelt es sich bei der Vererbung von `A` nach `B` um eine private Vererbung. Das Attribut `x` aus `A` ist nur von Methoden aus `B` zugreifbar. Außerhalb von `B` kann niemand auf `x` zugreifen, da `x` zum `A`-Anteil in `B` gehört, und dieser privat ist.

Listing 2.68. Öffentliche Vererbung

```
class A
{
public:
  int x;
};

class B : public A
{
};
```

Bei einer `public`-Vererbung ist der Anteil der Elternklasse in der Kindklasse öffentlich. Das heißt, dass von außerhalb auf das Element `x` in `B` zugegriffen werden kann.

Ebenso kann in C++ eine Vererbung „geschützt“ durchgeführt werden. Der Bezeichner `protected` führt dazu, dass die Basisklasse nur innerhalb der Namensräume der abgeleiteten Klassen sichtbar ist. Wie schon angedeutet wurde, sind die Möglichkeiten von C++ auf ältere Auffassungen über die OO-Programmierung zurückzuführen. Eine Bedeutung für OO-Konzepte besitzen diese Möglichkeiten daher nicht. Andererseits kann man immer wieder überrascht darüber sein, welche Tricks mit C++ möglich sind, spezielle Teilprobleme mit ganz unglaublichen Konstrukten zu lösen. Das mag man bejubeln oder beweinen, es ändert nichts an der Beschaffenheit von C++, das eben sehr vieles aus der Vergangenheit mitführt.

Der Konstruktor

Die Konstruktoren sind Methoden, die nur zur Initialisierung von Objekten definiert werden. Jede Instanziierung eines Objekts in C++ ruft einen Konstruktor auf. Konstruktoren können vom Programmierer definiert oder auch in bestimmten Fällen vom Compiler generiert werden. Die Regeln zum Umgang mit Konstruktoren sind sehr wichtig. Deshalb soll das Thema „Konstruktoren“ hier sehr ausführlich behandelt werden. Zunächst einmal zu einem kleinen Beispiel:

Listing 2.69. Ein Punkt in einem zweidimensionalen Koordinatensystem

```
struct Punkt
{
  int x
  int y;
};
```

Die Struktur `Punkt`, wie sie hier definiert ist, kennen wir schon aus vorangegangenen Abschnitten. Wenn eine Instanz dieser Struktur im Speicher angelegt wird, muss diese Instanz mit nachfolgenden Anweisungen initialisiert werden. Zur Initialisierung der Datenelemente kann in C++ ein Konstruktor definiert werden. Im folgenden Listing wird gleich die übliche Klassenschreibweise aus C++ verwendet.

Listing 2.70. Die Definition eines Konstruktors

```
// Klassendeklaration
class Punkt
{
public:
  Punkt( int vx, int vy );
  int x
```

```
  int y;
};

// Methodendefinition
Punkt::Punkt( int vx, int vy )
{
  x = vx;
  y = vy;
}

// Instanziierung eines Objekts
Punkt p1( 5, 5,);
```

Die zwei ersten Regeln zur Definition eines Konstruktors lauten:

1. *Der Konstruktor hat immer den Namen der Klasse.*
2. *Ein Konstruktor hat keinen Rückgabewert.*

Wenn diese beiden Regeln beachtet werden, kann man schon Konstruktoren definieren. Für die Definition gelten ansonsten alle Regeln, die man für Methoden anwendet. Ein Konstruktor kann beispielsweise inline definiert werden oder überladen werden. Wenn ein Konstruktor definiert wurde, muss er auch verwendet werden, außer er wurde durch weitere Konstruktoren überladen. Im letzteren Fall hat man die Wahlfreiheit zwischen den vorhandenen Konstruktoren. *Jede Instanziierung eines Objektes steht im Zusammenhang mit einem Konstruktoraufruf. Anders herum steht jeder Konstruktoraufruf im Zusammenhang mit der Instanziierung eines Objektes.* In den nachfolgenden Abschnitten werden die Themen behandelt, die im Zusammenhang mit der Definition von Konstruktoren stehen. Zunächst muss die Möglichkeit behandelt werden, Elemente von Klassen und Basisklassen zu initialisieren. Schließlich handelt es sich bei ihnen ebenfalls um Objekte.

Die Initialisierungsliste

Das Beispiel aus dem vorangegangenen Absatz kann etwas optimiert werden:

Listing 2.71. Eine Initialisierungsliste

```
// ...
Punkt::Punkt( int vx, int vy ) : x(vx), y(vy)
{
}
// ...
```

Statt die Klassenattribute innerhalb des Methodenblocks zu belegen, gibt es für Konstruktoren einen speziellen Platz, wo üblicherweise Elemente initialisiert werden: *die Initialisierungsliste*. Der Grund, warum dafür eine extra

syntaktische Regelung getroffen wurde, ist der folgende: Innerhalb des Methodenblocks können nur Variablen verändert werden, die schon eine Speicherstelle besitzen. Sie müssen also schon existieren und wurden eine kurze Zeit vorher instanziiert. Die Initialisierungsliste definiert den Zeitpunkt, an dem die Elemente wirklich instanziiert werden. Es findet also mit der Initialisierung keine Reinitialisierung statt, wie es im Methodenrumpf der Fall wäre. Syntaktisch findet die Initialisierung dadurch statt, dass der Name des Elements mit den entsprechenden Konstruktorparametern in die Initialisierungsliste geschrieben wird. Mehrere Initalisierungen werden durch Kommas getrennt. Wenn in der Initialisierungsliste kein Konstruktor für ein Datenelement explizit angegeben wird, wird dessen Standardkonstruktor verwendet. Auch Standarddatentypen haben pro forma einen Standardkonstruktor. Wenn es sich um Attribute eines Klassentyps ohne Standardkonstruktor handelt, ist der Code ohne explizite Initialisierungsliste nicht übersetzbar. Das folgende Beispiel kann aus dem genannten Grund nicht compiliert werden:

Listing 2.72. Fehlende Initialisierungsliste

```
class Punkt
{
public:
  Punkt( int vx, int vy ) { x = vx; y = vy; }
  int x;
  int y;
};

class Kreis
{
public:
  Kreis( const Punkt &vm, double vr )
  {
    m = vm;
    r = vr;
  }
  Punkt m; // m muss konstruiert werden!
  double r;
};
```

Die Initialisierungslisten der Klassen `Punkt` und `Kreis` sind leer und rufen deshalb die Standardkonstruktoren der Elemente auf. Die Standarddatentypen `int` und `double` haben Standardkonstruktoren. Lediglich die Klasse `Punkt`, die in `Kreis` für den Mittelpunkt verwendet wird, besitzt keinen Standardkonstruktor[41]. Der Konstruktor von `Kreis` benötigte also, um übersetzt

[41] Siehe Abschnitt 2.2.21 auf Seite 85.

werden zu können, eine explizite Initialisierungsliste. Auch für die Initialisierung der Elemente, die mit Standarddatentypen typisiert sind, wäre die Initialisierungsliste der richtige Platz. Es ist effizienter als die Reinitialisierung, die im Konstruktorrumpf stattfindet. Das Beispiel in der korrekten Variante:

Listing 2.73. Korrekte Initialisierungsliste

```
class Punkt
{
public:
  Punkt( int vx, int vy ) : x(vx), y(vy) {}
  int x;
  int y;
};

class Kreis
{
public:
  Kreis( const Punkt &vm, double vr )
  : m(vm.x,vm.y), r(vr) {}

  Punkt m;
  double r;
};
```

Ebenso wie die Elemente einer Klasse werden die Basisklassenanteile in der Initialisierungsliste initialisiert.

Listing 2.74. Initialisierung von Basisklassenelementen

```
class Form
{
public:
  Form( unsigned int farbe ) : farbwert(farbe) {}
protected:
  unsigned int farbwert;
};

class Kreis : public Form
{
public:
  Kreis( const Punkt &vm,
         double vr,
         unsigned int farbe )
  : Form(farbe), m(vm.x,vm.y), r(vr) {}
```

```
private:
  Punkt m;
  double r;
};
```

Auch sie werden einfach durch Kommas von den anderen Konstruktoraufrufen abgegrenzt. Um den Konstruktor einer Basisklasse zu bezeichnen, wird der Klassenname verwendet.

Ein sehr wichtiger und häufig missverstandener Punkt ist die Reihenfolge der Initialisierung der Elemente einer Klasse. Es ist nicht die Reihenfolge der aufgeführten Elemente in der Initialisierungsliste, sondern die der Klassendeklaration. Die Elemente werden in der Reihenfolge initialisiert, wie sie in der Klasse aufgeführt werden. Das ist auch der Grund dafür, warum man niemals Datenelemente der Klasse dazu verwenden sollte, innerhalb der Initialisierungsliste andere Klassenelemente zu initialisieren. Manche Compiler mahnen deshalb bei lesendem Zugriff von Klassenelementen in der Initialisierungsliste. Sie geben dann eine Warnung aus. Ebenso verursacht eine Reihenfolge der Elementinitialisierung in der Initialisierungsliste, die von der Deklarationsreihenfolge in der Klasse abweicht, bei vielen Compilern eine Warnung.

Konstante Objektattribute und Referenzattribute erfordern zwingend eine explizite Initialisierungsliste. Sie ist der einzige Platz, an dem ein konstantes Attribut initialisiert werden kann.

Listing 2.75. Initialisierung eines konstanten Attributs

```
class Bitmap
{
public:
  Bitmap() : kachelgroesse(4096) {}
private:
  const unsigned long kachelgroesse;
};
```

Dabei kann dieses Attribut für jedes Objekt einen anderen Wert annehmen, wenn es entsprechend initialisiert wird:

Listing 2.76. Initialisierung eines konstanten Attributs

```
class Bitmap
{
public:
  Bitmap( unsigned long g ) : kachelgroesse(g) {}

  unsigned long kachelGroesse() const
  { return kachelgroesse; }
private:
  const unsigned long kachelgroesse;
};
```

Für ein Objekt allerdings ist das Attribut `kachelgroesse` konstant.

Nur in wirklichen Ausnahmefällen kann es sinnvoll sein, ein Referenzattribut in eine Klasse aufzunehmen. Das folgende Beispiel soll zeigen, wie die Initialisierung von Referenzattributen ebenso auf die Initialisierungsliste angewiesen ist, wie die Initialisierung konstanter Attribute.

Listing 2.77. Initialisierung eines Referenzobjekts

```
class Logging
{
public:
  Logging( std::ostream _os ) : os(os) {}

  void schreibe( char *txt )
  {
    os << txt << std::endl;
  }

private:
  std::ostream &os;
};
```

Der Destruktor

Der Destruktor ist die Methode, die mit der Zerstörung des Objekts aufgerufen wird. Jedes Objekt, das im Speicher angelegt wird, muss auch aus dem Speicher entfernt werden. Bei diesem Vorgang wird immer der Destruktor aufgerufen. Aber nun zu den Regeln der Definition des Destruktors:

1. Wie der Konstruktor hat der Destruktor den Namen der Klasse. Allerdings wird dem Namen eine Tilde „~“ vorangestellt.
2. Der Destruktor hat eine leere Parameterliste. Er kann daher nicht überladen werden.
3. Er hat keinen Rückgabewert. Man deklariert auch nicht `void`!

Beispiel:

Listing 2.78. Deklaration eines Destruktors

```
class Punkt
{
public:
  Punkt( int vx, int vy );  // Konstruktor
  ~Punkt();                 // Destruktor
  int x
  int y;
};
```

Die Implementierung des Destruktors sieht folgendermaßen aus:

Listing 2.79. Definition eines Destruktors

```
Punkt::~Punkt()
{
}
```

Da im Falle des Punktes der Destruktor nicht wirklich etwas zu tun hat, ist der Methodenrumpf des Destruktors im gezeigten Beispiel leer. Bei Klassen, deren Objekte Systemressourcen beanspruchen, macht der Destruktor die nötige Aufräumarbeit. Systemressourcen werden alloziert, benutzt und danach dealloziert.

Listing 2.80. Speicherallokation im Konstruktor und Freigabe im Destruktor

```
typedef unsigned chat byte;

class Bitmap
{
  Bitmap( unsigned long s );
  ~Bitmap();
// ...
private:
  byte *buffer;
  unsigned long size;
};

Bitmap::Bitmap( unsigned long s ) : size(s)
{
  buffer = new[size];
}

Bitmap::~Bitmap()
{
  delete [] buffer;
}
```

Die Allokation übernimmt für gewöhnlich der Konstruktor und die Deallokation der Destruktor. Welche Syntax die beiden Operatoren `new` und `delete` besitzen, lesen Sie im Abschnitt 2.2.23 auf Seite 97. Wenn für eine Klasse kein Destruktor definiert wird, generiert der Compiler einen leeren Destruktor zu dieser Klasse hinzu.

Der Standardkonstruktor

Der Standardkonstruktor ist der Konstruktor mit der leeren Parameterliste. Er nimmt keine Werte für die Initialisierung seines Objekts entgegen. Dieser

Standardkonstruktor, der im Englischen „default constructor“ genannt wird, hat eine bestimmte syntaktische Bedeutung, die über die Bedeutung anderer Konstruktoren hinausgeht. Er wird zum Beispiel vom Compiler zu einem Typ hinzugeneriert, wenn kein anderer Konstruktor definiert wurde. Das stellt sicher, dass ein Typ instanziiert werden kann. Denn jede Instanziierung einer Klasse geht mit dem Aufruf eines Konstruktors einher. Außerdem wird der Standardkonstruktor vom Compiler auch zur Instanziierung genutzt, wenn kein anderer Konstruktor explizit genannt wird. Das folgende Beispiel zeigt einen solchen Fall:

Listing 2.81. Der Standardkonstruktor

```
#include <iostream>

class A
{
public:
  A() { std::cout << "A::A()" << std::endl; }
};

class B : public A
{
};

int main()
{
  B b;
  return 0;
}
```

Die Ausgabe auf der Konsole `A::A()` zeigt, dass der Standardkonstruktor von `A` bei der Instanziierung von `B` aufgerufen wurde. Mit welchen Mechanismen das geschieht, muss hier etwas genauer betrachtet werden. Der Standardkonstruktor in `A` kann nicht einfach nach `B` vererbt und als `B`-eigener akzeptiert werden. Er ist natürlich in `B` vorhanden und sichtbar, denn er wurde öffentlich deklariert. Er bleibt aber weiterhin ausschließlich ein Konstruktor für `A`. Eine Ausstattung abgeleiteter Klassen durch Konstruktoren der Basisklassen gibt es in C++ nicht – es ist mir auch keine Sprache bekannt, die so etwas zulässt. Der Aufruf muss also indirekt erfolgen. Dazu sehen wir uns die Klasse `B` an. Sie definiert überhaupt keinen Konstruktor. Damit tritt der Fall ein, dass der Compiler einen Standardkonstruktor generiert. Die Klasse `B` hat also einen Standardkonstruktor. Da wir keinen Konstruktor in `B` definiert haben, können wir auch keine Initialisierungsliste geschrieben haben, in der ein spezieller Konstruktor von `A` aufgerufen werden könnte. Dabei hätte uns die Klasse `A` selbstverständlich auch nur wenig zur Wahl gestellt. Neben dem Standard-

konstruktor gibt es nur den Kopierkonstruktor, wie im folgenden Abschnitt noch näher erklärt werden wird. Da also die Initialisierungsliste des generierten Standardkonstruktors von `B` keine andere Definition enthält, ruft sie den Standardkonstruktor von `A` auf. Es ist wichtig zu verstehen, dass es zwei unterschiedliche Regeln sind, die im Zusammenhang mit dem Standardkonstruktor stehen: Einerseits die Regel, die angibt, dass ein Standardkonstruktor durch den Compiler generiert wird, und andererseits die Regel, die den Aufruf des Standardkonstruktors vorschreibt, wenn kein anderer Konstruktor explizit aufgerufen wird.

Der Kopierkonstruktor

Der Kopierkonstruktor wird immer in der Situation aufgerufen, wenn eine Kopie eines Objektes instanziiert wird, d. h., wenn dem Konstruktor ein schon existierendes Objekt als Parameter übergeben wird. In seiner Definition gleicht er einem beliebigen anderen Konstruktor, bis auf seine spezielle Parameterliste, die folgendermaßen aussieht:

Listing 2.82. Der Copy-Konstruktor

```
class X
{
public:
  X() {} // Standardkonstruktor
  X( const X & ) {} // Copy-Konstruktor
};
```

Er bekommt als Parameter einfach eine Referenz des eigenen Klassentyps. Da in den meisten Fällen das Original bei einem Kopiervorgang nicht verändert wird, wird eine Referenz auf ein konstantes Objekt der eigenen Klasse übergeben. Man kann sich also die allgemeine Form `T::T(const T&)` merken. Es handelt sich aber auch dann um einen Kopierkonstruktor, wenn keine `const`-Maskierung des übergebenen Objekts vorgenommen wird. Es ist schließlich auch hier möglich, dass es für die Regel eine Ausnahme gibt und das Ursprungsobjekt beim Kopieren irgendeine Änderung erfahren muss. Also ist auch `T::T(T&)` ein Kopierkonstruktor. Man sollte sich aber soweit wie möglich an die erste Form halten.

Das Besondere an dem Kopierkonstruktor ergibt sich aus seiner herausragenden syntaktischen Stellung in C++. Er wird nämlich vom Compiler generiert, wenn er in einer Klasse nicht deklariert ist. Warum das so ist, soll in den nachfolgenden Listings gezeigt werden.

Listing 2.83. Verwendung des Copy-Konstruktors bei der Initialisierung

```
// ...

int main()
{
  X obj1; // Aufruf des Standardkonstruktors
  X obj2( obj1 ); // Aufruf des Copy-Konstruktors
  X obj3 = obj1;  // Noch einmal der Copy-Konstruktor

  return 0;
}
```

In dem gezeigten Code verhält es sich zunächst wie bei beliebigen Aufrufen beliebiger Konstruktoren. Es werden Objekte instanziiert und mit Vorgabewerten initialisiert. Wenn aber ein Objekt aus syntaktischen Gründen kopiert werden muss, hat man keinen explizit sichtbaren Konstruktoraufruf im Code, obwohl ein solcher durchgeführt werden muss.

Listing 2.84. Copy-Konstruktoraufruf bei Werteübergabe an eine Funktion

```
// ...

void f( X obj )
{
  // ...
}

int main()
{
  X obj1; // Standardkonstruktoraufruf

  f( obj1 ); // Aufruf des Copy-Konstruktors

  return 0;
}
```

Bei einer Werteübergabe der Parameter an eine Funktion[42] wird eine Kopie des entsprechenden Wertes oder des Objekts angelegt. Da eine Kopie eine ordentliche Instanz ist, muss sie auch durch einen Konstruktor erzeugt worden sein. Dafür wird der Kopierkonstruktor herangezogen. Damit nun Werteübergaben von Objekten ohne allzu großes Zutun des Programmierers durchgeführt werden können, garantiert der Compiler standardmäßig das Vorhandensein eines Kopierkonstruktors.

[42] Im Gegensatz zur Referenzübergabe von Parametern.

Das Problem mit dem Kopierkonstruktor

Aber warum ist es nun wichtig, diese Regel zu wissen? Das Problem beim generierten Kopierkonstruktor ist, dass der Compiler nichts über den semantischen Gehalt einer Klasse wissen kann. Der generierte Kopierkonstruktor macht einfach eine Bytekopie des existierenden Objekts. Aber gerade diese Bytekopie ist es, die bei vielen Klassen zu grobem Fehlverhalten führt, und zwar immer dann, wenn von dem kopierten Objekt ein weiteres Objekt abhängt und über einen Zeiger oder eine beliebige andere technische Indirektion[43] referenziert wird. In diesem Fall wird der Zeiger oder das Handle kopiert, das abhängige Objekt aber nicht. Das heißt, dass nun beide Objekte auf ein abhängiges Objekt verweisen. Üblicherweise werden diese abhängigen Objekte dann auch im Destruktor gelöscht. Und genau dies geschieht nun zweimal, was zu einem Laufzeitfehler führt. Der folgende Code führt zu einem solchen Laufzeitfehler:

Listing 2.85. Typischer Fehler im Zusammenhang mit dem Copy-Konstruktor

```
#include <cstdio>

class Datei
{
public:
  Datei( char *name )
  {
    f = std::fopen( name, "wt" );
  }
  ~Datei() { if( f ) fclose( f ); }

  void schreibe( char *txt )
  {
    if( f )
      fprintf( f, txt );
  }
private:
  std::FILE *f;
};

// ...

void f( Datei d )
{
```

[43] Ein Handle beispielsweise. Ein Objekt könnte ein Handle enthalten, das meinetwegen auf eine Datei verweist.

```
  d.schreibe( "Eine Zeile Text.\n" );
}

int main()
{
  Datei d( "test.txt" );

  f( d );
  d.schreibe( "Und noch eine Zeile\n" );

  return 0;
}
```

Je nach dem Verhalten des eingesetzten Betriebssystems erhält man einen Programmabsturz, oder eben ein unvollständiges Ergebnis. Beim Aufruf der Funktion `f()` wird eine Kopie des Objektes von der Klasse `Datei` erzeugt und das Filehandle eins zu eins kopiert. Dabei wird natürlich keine neue Datei geöffnet, sondern die Kopie arbeitet mit dem Filehandle des Originalobjekts weiter. Am Ende der Funktion `f()` wird die Kopie gelöscht. Mit dem Destruktoraufruf der Kopie wird auch die Datei geschlossen. Das gleiche Problem tritt auf, wenn Speicher für ein Objekt allokiert wurde und ein kopiertes Objekt nun ein Duplikat des Zeigers enthält. Alles nur, da der vom Compiler generierte Kopierkonstruktor eine Bytekopie anfertigt.

Das Tückische dabei ist nun, dass der Compiler noch nicht einmal eine Warnung ausgibt, wenn er einen Kopierkonstruktor generiert und auch verwendet. Das alles geschieht völlig ohne eine Rückmeldung für den Entwickler. Genau das ist auch das eigentliche Problem. Das Verhalten des Compilers in diesen Situationen wird häufig schlichtweg vergessen. Ich kann aus meiner Erfahrung als C++-Entwickler sagen, dass ich kaum ein C++-Projekt erlebt habe, in dem dieser Fehler der Nichtbeachtung des Kopierkonstruktors nicht wenigstens einmal gemacht wurde. Meistens trat der Fehler auf, wenn ein Entwickler eine Klasse programmierte, die ein anderer Entwickler benutzen sollte. Irgendwann wurde dann einmal das Zeichen für die Referenz in einer Parameterliste vergessen und schon gab es eine Objektkopie mit den beschriebenen Folgen. Die meisten Projekte haben Codingstyles, in denen steht, dass der Kopierkonstruktor für jede Klasse geschrieben werden soll. Aber wie verhindert man nun ein unbeabsichtigtes Kopieren? Der folgende Code zeigt, wie man den Kopierkonstruktor für eine Klasse versteckt:

Listing 2.86. Privater Copy-Konstruktor

```
class Datei
{
public:
  Datei( char *name )
```

```
  {
    f = std::fopen( name, "wt" );
  }
  ~Datei() { if( f ) fclose( f ); }

  void schreibe( char *txt )
  {
    if( f )
      fprintf( f, txt );
  }
private:
  std::FILE *f;

  // in privater Sektion verstecken!
  Datei( const Datei & );
};
```

Der Kopierkonstruktor wird einfach in der privaten Sektion der Klasse deklariert. Man sollte ihn nicht definieren, da er ja dann innerhalb von Klassenmethoden wieder zur Verfügung steht. Wenn nun eine Wertekopie erzeugt werden soll, gibt der Compiler eine Fehlermeldung aus, denn auf den Kopierkonstruktor kann nicht zugegriffen werden. Damit wird das unbeabsichtigte Kopieren verhindert. Mit anderen Worten: Es wird ein semantischer Fehler zu einem syntaktischen gemacht.

Der andere Fall ist nun, dass eine Kopie eines Klassenobjektes möglich sein soll. Dafür muss dann ein Kopierkonstruktor definiert werden, der die nötigen algorithmischen Voraussetzungen mitbringt, entweder ein weiteres abhängiges Objekt für die Kopie zu erzeugen, oder der eine Logik implementiert, wie ein abhängiges Objekt zwischen Original und Kopie geteilt werden kann.

2.2.22 Typenkonvertierung mit Konstruktoren

Wenn man sich genau durchdenkt, was eigentlich bei einer Typenkonvertierung geschieht, so gelangt man dazu, dass eine Art Kopiervorgang stattfindet, bei dem das Ziel eine andere Organisationsform besitzt als die Quelle. Auf jeden Fall wird die Quelle der Daten in ihrer Struktur nicht verändert. Es muss also etwas Neues entstehen. Das ist auch der Grund, warum bei Typenkonvertierungen von Klassentypen Konstruktoren mit von der Partie sind. Es sind die Konstruktoren, die einen Parameter haben, die zur Typenkonvertierung herangezogen werden können.

Weil es so anschaulich ist, werde ich auch die folgenden Beispiele im zweidimensionalen Raum abspielen lassen. Stellen wir uns also Punkte und

Vektoren in der Ebene vor und lehnen wir das nächste Beispiel an die Lineare Algebra an. Punkte haben ihre Koordinatenpaare, Vektoren haben ebenfalls zwei Koordinaten, die allerdings Deltas darstellen. Ein Vektor hat keinen Ort, er beschreibt eine Kraft oder eine Wegstrecke in Form von Richtung und Länge. Einen Punkt kann man also auch als einen Vektor ansehen, der am Nullpunkt ansetzt und diesen verschiebt – einen Ortsvektor. Diese Überlegungen sollen in dem Beispiel realisiert werden:

Listing 2.87. Implizite Typenkonvertierung mit dem Konstruktor

```
#include <iostream>

class Punkt
{
public:
  Punkt( int vx, int vy ) : x(vx), y(vy) {}
  Punkt( const class Vektor &v );

  int x;
  int y;
};

class Vektor
{
public:
  Vektor( int vx, int vy ) : dx(vx), dy(vy) {}

  int dx;
  int dy;
};

Punkt::Punkt( const Vektor &v ) : x(v.dx), y(v.dy) {}

int main()
{
  Vektor v( 3, 4 );
  Punkt p( 0, 0 );

  p = v; // implizite Typenkonvertierung!

  return 0;
}
```

Dadurch, dass die Klasse `Punkt` einen Konstruktor hat, der eine Referenz auf einen `Vektor` entgegennimmt, kann in der Funktion `main()` die Zuweisung eines `Vektor`-Objekts zu einem `Punkt`-Objekt stattfinden. Die rechte Seite muss zuerst vom Typ `Vektor` nach `Punkt` konvertiert werden. Danach kann zugewiesen werden. Die Konvertierung geschieht durch den besagten Konstruktor. Der Compiler organisiert den Code so, dass ein unbenanntes temporäres Objekt auf dem Stack angelegt wird, das mindestens so lange existiert, bis die Zuweisung stattgefunden hat.

Der Compiler bezieht also bei seiner Suche nach Möglichkeiten der Typenangleichung auch die einparametrigen Konstruktoren mit ein und verwendet sie bei Bedarf. Es gibt Situationen, in denen solche Eigenmächtigkeiten des Compilers aus fachlichen Gründen unterbunden werden müssen. Dazu kam im ANSI/ISO-C++ das Schlüsselwort `explicit` hinzu. Wendet man dieses Schlüsselwort auf einen Konstruktor an, so wird dieser nicht mehr „implizit" zur Typenkonvertierung herangezogen. Wenn also im Beispiel der besagte Konstruktor von Punkt folgendermaßen deklariert worden wäre:

Listing 2.88. `explicit`-Deklaration eines Konstruktors

```
class Punkt
{
public:
  // ...
  explicit Punkt( const class Vektor &v );
  // ...
};
```

hätte die implizite Konvertierung nicht stattfinden können. In diesem Fall hat man immer noch die Möglichkeit, die Konvertierung explizit durchzuführen.

Listing 2.89. Unterbundene implizite Konvertierung

```
p = v; // Ungültig! Implizite Konvertierung geht nicht.
p = Punkt(v); // Explizite Typenkonvertierung!
```

Damit wird auch sichtbar, was der Compiler im impliziten Fall automatisch einfügte. Es lässt sich auch erahnen, warum bei einer Typenkonvertierung in C++ die Klammern nicht um den Typ, sondern um das zu konvertierende Objekt gemacht werden sollten. Es ist die so genannte Konstruktorschreibweise. Schließlich wird dabei ja auch ein Konstruktor aufgerufen. Die alte C-Schreibweise wird übrigens auch in einen Konstruktoraufruf umgesetzt. Da die alte Schreibweise aber nicht so gut darstellt, was sie im Kern ist, ist es besser die Konstruktorschreibweise zu verwenden. Im ANSI/ISO-Standard ist für die explizite Konvertierung zwischen Typen mit definiertem Konvertierungsweg – also entsprechender Konstruktor oder Typenkonvertierungsoperator – das Schlüsselwort `static_cast<>()` vorgesehen. Im Beispiel sind also die drei Varianten der Konvertierung eines Vektors in einen Punkt gleichbedeutend:

1. `p = (Punkt)v;` Der „alte“ C-Cast.
2. `p = Punkt(v);` Die C++-Konstruktorschreibweise.
3. `p = static_cast<Punkt>(v);` Der neue C++-Konvertierungsoperator nach dem ANSI/ISO-Standard.

Der Sinn des neuen Operators liegt darin, dass man mit ihm kenntlich machen kann, welche Art der Konvertierung man im konkreten Fall meint. Der Operator `static_cast<>()` verlangt einen definierten Weg der Konvertierung in den Beschreibungen der beteiligten Typen. Damit grenzt er sich von den drei anderen Konvertierungsoperatoren `const_cast<>()`, `reinterpret_cast<>()` und `dynamic_cast<>()` ab[44]. Die Verwendung von `static_cast<>()` in den dafür vorgesehenen Fällen ist also empfehlenswert, da das an der entsprechenden Stelle etwas über die Intention des Entwicklers aussagt.

Um dem Abschnitt 2.2.31 ab Seite 127 etwas vorzugreifen: den Operator `static_cast<>()` wendet man überall dort an, wo der Konvertierungsweg in der Beschreibung der beteiligten Typen definiert ist. Einen `double`-Wert kann man mit `static_cast<>()` nach `int` konvertieren, weil das in C++ vordefiniert ist. Ein `char`-Array kann man damit nicht in ein `Punkt`-Array konvertieren, weil das nicht definiert ist.

2.2.23 Globale, automatische und dynamische Instanziierung

Die globale, die automatische und die dynamische Instanziierung sind die drei Möglichkeiten, Objekte von Klassen zu erschaffen. Die globale und die automatische Instanziierung geschehen einfach dadurch, dass Deklarationsanweisungen der Form `Typ Bezeichner;` angewendet werden. So ist beispielsweise `int i;` eine automatische oder globale Deklaration, je nachdem in welchem Gültigkeitsbereich die Deklaration steht. Auf dem globalen Gültigkeitsbereich oder als statische Klassenelemente spricht man von globaler Instanziierung, da die Objekte im Heap angelegt werden und ihr Bestehen den Durchlauf der `main()`-Funktion überdauert. Wenn Objekte per Deklaration auf einem lokalen Funktionsscope oder in einem noch tiefer verschachtelten Block angelegt werden, spricht man von automatischer Instanziierung. Solche Objekte werden auf dem Stack angelegt und von dort zur Laufzeit, beim Verlassen des entsprechenden Blocks auch wieder abgebaut.

Der Stack ist ein Speicherbereich, der zu Beginn der Programmausführung reserviert wird, um lokale (automatische) Daten aufzunehmen, die Parameterübergabe an Funktionen und schließlich auch den Rücksprung der Funktionen an die aufrufende Stellen zu ermöglichen. Den Namen hat der Stack – Stapel – durch seine Organisationsform. Es werden Daten wie auf ei-

[44] Siehe Abschnitt 2.2.31 auf Seite 127.

nem Stapel abgelegt und auch so wieder heruntergenommen. Das heißt, dass die obersten, zuletzt abgelegten Daten zuerst abgebaut werden. Auf modernen Desktop- und Serversystemen werden üblicherweise dynamische Stacks eingesetzt. Benötigt ein Programm sehr viele automatische Daten oder ist die Aufrufreihenfolge der Funktionen sehr tief verschachtelt und stößt dabei an die Grenzen des Stacks, so alloziert die Laufzeitumgebung einen größeren Stack. In Embedded-Systemen ist der Stack häufig statisch. Er hat dann oft eine Größe von 4, 8 oder 16K. Das ist nicht sehr viel. Wenn man auf solchen Systemen zu viel automatische Daten alloziert, werden die Grenzen des Stacks gesprengt und man bekommt einen Fehler zur Laufzeit. Es ist also nicht ratsam, größere Arrays im lokalen Funktionsscope anzulegen.

In C und C++ gibt es zur Bezeichnung der automatischen Instanziierung das Schlüsselwort `auto`, das aber nicht angewendet werden muss, da auf lokalen Scopes sowieso nur automatisch instanziiert werden kann. In C++ werden die globalen und automatischen Objekte alle mit ihrem Konstruktor initialisiert. Wenn ein Typ keinen Konstruktor besitzt, wird ein Standardkontsruktor generiert, der nur die eine Funktion besitzt, die jeweiligen Standardkonstruktoren der Elemente und der Basisklassen aufzurufen. Beim Zeitpunkt der Objektzerstörung, am Ende des Blocks bei automatischen Objekten und am Ende der Programmlaufzeit bei globalen Objekten werden die Destruktoren aufgerufen. Wenn ein Typ keinen Destruktor besitzt, wird auch dieser durch den Compiler hinzugeneriert.

Listing 2.90. Automatische Instanziierungen

```
#include <iostream>
using namespace std;
class X
{
public:
  X() { cout << "X::X()" << endl; }
  ~X() { cout << "X::~X()" << endl; }
};

void f()
{
  X x; // automatische Instanziierung
  cout << "Anfang Block innerhalb f()" << endl;
  {
    X x; // automatische Instanziierung
    // ...
  }
  cout << "Ende Block innerhalb f()" << endl;
}
```

```
int main()
{
  auto X x; // automatische Instanziierung
            // das Schlüsselwort auto ist optional
  cout << "Aufruf von f()" << endl;
  f();
  cout << "Nach dem Aufruf von f()" << endl;
  return 0;
}
```

Die Ausgabe des vorgestellten Beispiels ist die folgende:

```
X::X()
Aufruf von f()
X::X()
Anfang Block innerhalb f()
X::X()
X::~X()
Ende Block innerhalb f()
X::~X()
Nach dem Aufruf von f()
X::~X()
```

Diese Ausgabe veranschaulicht, dass die Konstruktoren bei der Deklaration und die Destruktoren am Ende des Blocks aufgerufen werden, in dem die Objekte angelegt wurden.

Auf dem globalen Scope ist die Reihenfolge der Instanziierung der Objekte undefiniert. Ebenso ist die Reihenfolge der Zerstörung nicht festgelegt. Das zu beachten ist besonders wichtig, denn es könnte der Destruktor einer Klasse auf weitere globale Objekte zugreifen wollen, wie in dem eben gezeigten Beispiel. Dort greift der Destruktor auf das Objekt `cout` zu. Wenn nun ein Objekt einer solchen Klasse global instanziiert wird, die in ihrem Konstruktor oder Destruktor andere globale Objekte benötigt, greift die Instanz womöglich ins Leere. Aus diesem Grund habe ich in dem Beispiel keine globale Instanziierung eines Objektes der Klasse `X` vorgenommen. Das wäre ein schwerer Fehler, der je nach verwendetem Compiler unentdeckt bleiben oder zu einem Laufzeitfehler führen würde. Globale Instanziierungen können also fehlschlagen, wenn Objekte im Konstruktor oder Destruktor interagieren. Objekte, die nur einfache Initialisierungen haben, können problemlos global instanziiert werden.

Die dritte Form der Instanziierung von Objekten ist die dynamische. Die Objekte, die auf diese Art entstehen, sind unabhängig von irgendwelchen Blockgrenzen oder Gültigkeitsbereichen. Auch die Zeitpunkte der Instanziierung und der Zerstörung sind frei wählbar. Und genau darin liegt der Sinn dynamisch angelegter Objekte. Sie sind vom Rest des Programms und des

Stackzustands unabhängig. Instanziiert werden diese Objekte mit dem Operator `new`, gelöscht werden sie mit `delete`.

Die Operatoren `new` und `delete`

Die Instanziierung von Objekten mit dem Operator `new` kann zu jeder beliebigen Zeit des Programmablaufs geschehen[45]. Dazu wird nach `new` der Typ bzw. der Konstruktor des Typs aufgeführt, der instanziiert werden soll. Dabei reicht `new` einen richtig typisierten Zeiger auf das angelegte Objekt zurück. Im Fehlerfall reagiert `new` in ANSI/ISO-C++ mit einer Exception. In älteren AT&T-Varianten oder solchen, die an den AT&T-Standard angelehnt sind, liefert `new` im Fehlerfall einen Nullzeiger zurück. Diese unterschiedlichen Verhaltensweisen zwischen alten und neuen C++-Varianten sind für manches Integrationsproblem zwischen alten und neuen Codes verantwortlich.

Ein Beispiel mit ANSI/ISO-konformer Instanziierung:

Listing 2.91. Dynamische Instanziierung

```
#include <iostream>

class X
{
public:
  X() { std::cout << "X::X()" << std::endl; }
  ~X() { std::cout << "X::~X()" << std::endl; }
};

int main()
{
  try
  {
    X *p = new X;

    delete p;
  }
  catch( std::bad_alloc &e )
  {
    // Fehlerfall
  }

  return 0;
}
```

[45] Sogar vor und nach der Ausführung von `main()`.

Das gleiche Beispiel nach dem AT&T-Standard zeigt den Unterschied:

Listing 2.92. Dynamische Instanziierung nach AT&T-Standard

```
#include <iostream.h>

class X
{
public:
  X() { cout << "X::X()" << endl; }
  ~X() { cout << "X::~X()" << endl; }
};

int main()
{
  X *p = new X;
  if( p != NULL )
  {

    delete p;
  }
  else
  {
    // Fehlerfall
  }

  return 0;
}
```

In beiden Fällen bedeutet die reine Nennung des Typs nach dem Operator `new`, dass dieser den Standardkonstruktor der Klasse verwendet. Ebenso kann ein Konstruktor explizit genannt werden: `X *p = new X();` nennt den Standardkonstruktor explizit. Wenn die Klasse weitere Konstruktoren besitzt, können diese durch die Übergabe geeigneter Parameter genannt werden. Eine Instanz des Typs Punkt aus dem Abschnitt 2.2.21 über Konstruktoren könnte folgendermaßen instanziiert werden:

```
Punkt *p = new Punkt(1,1);
```

Damit wird der Konstruktor mit zwei ganzzahligen Parametern zur Initialisierung verwendet. Es wurde die alte und die neue Fehlerbehandlung des Operators `new` gezeigt. Dabei gibt es auch in ANSI/ISO-C++ noch die Möglichkeit, den Operator zur alten Verhaltensweise zu bewegen, der Nullpointerrückgabe im Fehlerfall. Damit befasst sich der Abschnitt 2.2.23 auf Seite 99.

Die Zeiger auf dynamisch erzeugte Objekte müssen gut aufbewahrt werden. Denn nur über sie kann auf die Objekte zugegriffen werden. Auch für das

Löschen sind die Zeiger wichtig. Verliert man alle Zeiger auf ein dynamisch angelegtes Objekt, kann dieses nicht mehr aus dem Speicher entfernt werden. Und genau darauf ist peinlichst zu achten, dass alle mit `new` angelegten Objekte auch wieder aus dem Speicher entfernt werden. Das geschieht mit dem Operator `delete`, dem Gegenstück zu `new`. Dabei muss man `delete` lediglich den Zeiger übergeben. Wie in den vorangegangenen Beispielen gezeigt, steht ein Zeiger auf ein dynamisch angelegtes Objekt einfach hinter dem Operator `delete`. Am Typ des Objekts erkennt der Operator, welcher Destruktor aufgerufen werden soll. Ein Destruktor wird praktisch – fast – nie direkt aufgerufen. Bis auf eine sehr spezielle Ausnahme geschieht dies automatisch oder durch den `delete`-Operator.

Das Paar `new` - `delete` verhält sich gegenüber `malloc()` und `free()` dadurch unterschiedlich, dass Speicher nicht einfach alloziert und wieder freigegeben wird, sondern immer eine Initialisierung und eine Deinitialisierung stattfindet.

Eine Sonderform des Paars `new` und `delete` ist die, die für die Allokation von Arrays verwendet werden kann. Dabei steht hinter dem Typ die Größe des Arrays in eckigen Klammern. Also `new Typ[x]`, wobei x für die Größe des Arrays steht. In dem Fall werden alle Elemente des Arrays mit dem Standardkonstruktor initialisiert. Dass die beiden Varianten sehr ähnlich aussehen können, führt manchmal zu Fehlern. Deshalb ist Vorsicht geboten. Die Variante `new T[n]` legt ein Array von `n` Objekten des Typs `T` an. Die Variante `new T(n)` legt ein Objekt des Typs `T` an und initialisiert es mit dem Wert `n`. Weiter ist darauf zu achten, dass der korrekte `delete`-Operator verwendet wird. Wenn ein Array von Objekten angelegt wurde, muss dieses durch `delete []` freigegeben werden. Also `delete [] p`, wenn p ein Zeiger auf ein mit `new` angelegtes Array ist. Wird der `delete`-Operator ohne die eckigen Klammern auf ein solches Array angewendet, wird nur das erste Objekt herausgelöscht. Es ist nicht möglich die Elemente von dynamisch allozierten Arrays durch einen anderen Konstruktor als den Standardkonstruktor zu initialisieren!

Die Nothrow - Variante des Operators new

In den vorangegangenen Abschnitten wurde der Operator `new` in seiner ASNI/ISO-konformen Funktionsweise und in der des AT&T-Standards vorgestellt. Dabei wirft der aktuelle `new`-Operator bei einem Fehlschlag der Allokation die Exception `bad_alloc`. Die ältere Variante gibt einen Nullzeiger zurück. Wenn nun ältere Codes zu standardkonformem Code portiert werden sollen, muss die alte Verhaltensweise auch mit einem Standardcompiler anwendbar sein. Für diesen Fall gibt es auch in ANSI/ISO-C++ eine Variante des `new`-Operators, die einen Nullzeiger im Fehlerfall zurückgibt. Diese Variante muss mit dem Markerinterface `std::nothrow` gekennzeichnet werden. Das folgende Beispiel gleicht strukturell dem AT&T-Beispiel. Es ist allerdings konform zum ANSI/ISO-Standard.

Listing 2.93. Die `nothrow`-Variante des `new`-Operators

```
#include <iostream>
using namespace std;

class X
{
public:
  X() { cout << "X::X()" << endl; }
  ~X() { cout << "X::~X()" << endl; }
};

int main()
{
  X *p = new(nothrow) X;
  if( p != NULL )
  {

    delete p;
  }
  else
  {
  }

  return 0;
}
```

Der Bezeichner `nothrow` bezeichnet einfach einen leeren Typ, der nur dazu verwendet wird, den speziellen `new`-Operator von dem zu unterscheiden, der eine Exception wirft. Wie alle Elemente der Standardbibliothek befindet er sich im Namensraum `std`.

2.2.24 Speicherklassen

C++ kennt sieben Schlüsselwörter um sogenannte „Speicherklassen“ zu spezifizieren. Das bedeutet, dass die genannten Schlüsselwörten Einfluss auf die Art oder den Ort des Speicheraufenthalts eines Objekts haben. An dieser Stelle sollen die Speicherklassen kurz aufgelistet werden, um in weiteren Abschnitten genauer erklärt zu werden.

```
auto
register
extern
static
```

```
mutable
const
volatile
```

Manche dieser Schlüsselwörter können kombiniert werden, andere nicht. So ist zum Beispiel `const` und `mutable` ein Gegensatzpaar, ebenso wie `auto` und `register`.

`auto`

Das Schlüsselwort `auto` existiert zwar in C++. Seine Verwendung ist jedoch unnötig, denn jede lokale Variablendeklaration oder Objektinstanziierung hat die Speicherklasse `auto`, wenn nichts anderes spezifiziert ist. Die Deklaration

```
int i;
```

kann ebenso mit

```
auto int i;
```

geschrieben werden. Die Speicherklasse besagt, dass eine Variable in einem lokalen Gültigkeitsbereich auf dem Stack angelegt wird und beim Verlassen dieses Bereichs automatisch vom Stack entfernt wird. Der Speicherort ist also der Stack.

Ein Grund für die Existenz des Schlüsselwortes scheint mir eine angestrebte Vollständigkeit zu sein. Ein anderer Grund könnte sein, dass mit `auto` einem optimierenden Compiler explizit gesagt werden soll, dass eine Variable eben nicht in einem Register landen darf.

`register`

Die Speicherklasse `register` grenzt sich von `auto` ab, indem man damit eine Empfehlung an den Compiler abgibt, die Variable in einem Prozessorregister zu platzieren. Insbesondere auf älteren Prozessorarchitekturen konnte man mit dieser Speicherklasse einiges an Laufzeitreduktion erreichen. Für die Intelarchitekturen waren das die Prozessorgenerationen bis zum 386er. Diese alten Prozessoren arbeiteten alle Maschinenbefehle sequenziell ab. Wenn Berechnungen durchgeführt wurden, die auf vorangegangenen beruhten, war es eine Optimierungsstrategie, möglichst viele Lade-und Entladeoperationen (`MOV`) dadurch zu sparen, dass man die Berechnungsergebnisse in den Prozessorregistern beließ. Der C- und C++-Programmierer konnte darauf Einfluss nehmen, indem er die Speicherklasse `register` für Variablen wählte, die besonders häufig verändert wurden. Die folgende Variable könnte vielleicht eine Schleifenvariable sein.

```
register int i;
```

Da Prozessoren keine unendliche Anzahl von Registern besitzen und darüber hinaus die Anzahl der Register architekturabhängig ist, kann eine „Anfrage" nach dieser Speicherklasse nicht in jedem Fall erfüllt werden. Die Speicherklasse `register` ist also eine Empfehlung, die vom Compiler nicht unbedingt erfüllt werden muss.

Mit der Einführung der Pipe-Struktur bei den Mikroprozessoren[46] wurde eine Parallelität in der Verarbeitung eingeführt, die die oben genannte Optimierungsstrategie wertlos machte. Der Prozessor kann Befehle parallel bearbeiten, die unabhängig voneinander sind. Diese Befehle dürfen also gerade nicht Ergebnisse von direkt vorangegangenen Operationen benötigen. In dem Maß, wie heute Compiler Optimierungen verfolgen, die auf die Ausnutzung der Pipe-Struktur zielen, ist die Verwendung der Speicherklasse `register` sinnlos geworden.

Natürlich gibt es auch heute noch kleine Prozessoren in Embedded-Systemen, die noch nach dem alten Prinzip funktionieren und keine Parallelverarbeitung unterstützen. In Projekten für solche Prozessoren kann diese Speicherklasse weiterhin sinnvoll sein.

`extern`

Die Speicherklasse `extern` bezeichnet Symbole[47], die in anderen Übersetzungseinheiten zu finden sind. Globale Daten müssen mit `extern` deklariert werden, wenn der Compiler sie nicht in der aktuellen Übersetzungseinheit anlegen soll. Solche Daten haben dann eine so genannte externe Bindung. Funktionsprototypen sind standardmäßig `extern`, auch wenn man sie nicht explizit als solche deklariert. Sie haben die externe Bindung also per Voreinstellung. Externe Symbole sind immer global.

`static`

Etwas schwieriger verhält es sich mit der Speicherklasse `static`. An globalen Daten angebracht, bewirkt diese Speicherklasse die „Modul-Lokalität". Eine Variable `x`, die in dem globalen Namensraum mit `static int x;` deklariert wird, ist modul-lokal. Das heißt, dass diese Variable `x` nicht mit einer anderen Variablen `x` in Konflikt gerät, die in einem anderen Modul deklariert wurde. Diese Variable untersteht damit der (Modul-)internen Bindung. Damit grenzt die Speicherklasse `static` Symbole mit interner Bindung von solchen mit externer Bindung ab. Auch Funktionen können mit interner Bindung deklariert

[46] Die erste Pipe-Struktur der Intel-Prozessoren wurde mit dem 486er eingeführt. Mit dem Pentium wurde das Verfahren perfektioniert, sodass der eigentliche Geschwindigkeitssprung durch diese neue Technik erreicht wurde.

[47] Symbole sind die Elemente, die der Linker in seinen Objektdateien finden kann. Diese Symbole haben Namen, die aus den Bezeichnern in C und C++ gebildet sind. Das „Namemangling" unterscheidet sich dabei in den beiden Sprachen. Nicht jeder Bezeichner führt gleich zu einem Symbol in der Objektdatei. Voraussetzung dafür ist, dass das bezeichnete Element Speicher benötigt. Es sind also Funktionen, Methoden und globale Daten.

werden, indem man sie mit dem Schlüsselwort `static` deklariert. Manche Compiler haben bei Funktionen mit interner Bindung eine bessere Möglichkeit zur Optimierung.

Wenn man lokale Daten statisch deklariert, findet eine Ausweitung der Lebensdauer der Daten statt. Die Daten haben dann eine „globale" Lebensdauer mit der einen Einschränkung, dass eine Initialisierung der Daten erst beim ersten Funktionsdurchlauf stattfindet. Die Bezeichner der Daten sind dabei nur im lokalen Block zugreifbar.

Listing 2.94. Statische Variable in einer Funktion

```
unsigned int globaler_zaehler()
{
  // Wird nur einmal initialisiert!
  static unsigned int z = 0;

  // Wird bei jedem Funktionsdurchlauf inkrementiert.
  z++;

  return z;
}
```

Eine etwas andere Bedeutung bekommt das Schlüsselwort static, wenn es an Klassenelementen angebracht wird. Es wird damit zum Ausdruck gebracht, dass das entsprechende Element kein Objektattribut ist. Es wird also nicht mit jeder Objektinstanziierung vervielfältigt, sondern besteht für alle Objekte der entsprechenden Klasse genau einmal. Die Methoden der Klasse können auf statische Elemente zugreifen. Statische Methoden können nicht auf Objektattribute zugreifen. Statische Elemente von Klassen kann man auch anhand des Klassennamens verwenden, ohne dass die Instanziierung eines Objekts nötig wäre. In diesem Zusammenhang führt das Schlüsselwort `static` zur externen Bindung des Symbols.

`mutable`

Eine etwas gewöhnungsbedürftige Speicherklasse ist `mutable`. Sie hebt die Eigenschaft der Unveränderlichkeit bei Elementen konstanter Objekte auf. Wird ein Element mit `mutable` spezifiziert, kann es auch dann verändert werden, wenn das Objekt selbst konstant ist. Also:

Listing 2.95. Ein `mutable`-Attribut

```
class X
{
  public:
    X() : a(0), b(0) {}
```

```
    void f() const
    {
      a = 0; // Nicht erlaubt!
      b = 0; // OK!
    }

    int a;
    mutable int b;
};

int main()
{
  const X obj;

  obj.a = 0; // Nicht erlaubt!
  obj.b = 0; // OK!

  return 0;
}
```

Begründen kann man das Vorhandensein dieser Speicherklasse mit bestimmten Fällen, in denen eine logische Unveränderlichkeit nicht gleichzeitig eine physikalische sein muss. Das kann dann der Fall sein, wenn ein für den logischen Zusammenhang ganz unabhängiger Aspekt ins Spiel kommt. So könnte ein lesender Objektzugriff auf Attribute eines Objekts durch Synchronisationsmechanismen geschützt werden müssen, wenn man das Objekt in einem Umfeld verwendet, in dem von mehreren Threads[48] darauf zugegriffen wurd. Das Synchronisationsobjekt selbst könnte Bestandteil des fachlichen Objekts sein. Die Synchronisation besteht darin, dass das Synchronisationsobjekt verändert wird. Natürlich lässt sich dieser Fall auch anders lösen ohne das Schlüsselwort `mutable` zu verwenden. Überhaupt ist von einer intensiven Verwendung dieser Speicherklasse abzuraten. Es wird eher dazu verwendet, Fehler im Design nachträglich zu korrigieren. In diesem Sinne hat es eine Verwandschaft zu `const_cast<>()`. Auch dieser Cast entfernt die logische Eigenschaft der Unveränderlichkeit. Wenn diese logische Unveränderlichkeit mit einer physikalischen zusammenfällt, muss bei der Verwendung von `const_cast<>()` ein Laufzeitfehler auftreten.

Die Speicherklasse `mutable` wird bis jetzt noch nicht von jedem Compiler unterstützt.

[48] Siehe Abschnitt 6.3.2 auf Seite 382.

`const`

Mit `const` deklariert man Konstanten. Im Gegensatz zu den mit `#define` definierten Textkonstanten sind die mit `const` deklarierten Konstanten typisiert. Der Compiler erlaubt damit keinen schreibenden Zugriff auf den entsprechenden Speicher[49], weshalb der initiale Wert bei einer Konstantendeklaration direkt genannt werden muss.

```
const int maximale_hoehe = 10000;
```

Das Schlüsselwort `const` hat noch andere Aufgaben. Es „maskiert" in Parameterlisten Objekte als konstant, obwohl die dahinter liegenden Objekte durchaus variabel sein können. Auf diesen Aspekt und auf weitere wird in Abschnitt 2.2.34 ab Seite 141 genauer eingegangen.

`volatile`

In bestimmten Situationen kann es nötig sein, die Lade- und Entladeoptimierung beim Variablenzugriff zu unterdrücken. Das ist immer der Fall, wenn es um parallelen Variablenzugriff aus separaten Ablaufsträngen in der Software geht; also in Programmen mit mehreren Threads oder in der Interruptprogrammierung. Wenn separate Ablaufeinheiten auf eine Variable zugreifen, diese aber in einem Register zwischengespeichert werden würde, dann kann es sein, dass die Zugriffe auf unterschiedliche Werte erfolgen. Die Speicherklasse `volatile` zwingt den Compiler dazu, die Optimierung zu unterdrücken und Werte immer aus dem Speicher zu lesen, beziehungsweise in den Speicher zu schreiben. So könnte etwa der folgende Codeabschnitt durch den Compiler optimiert werden:

Listing 2.96. Schleife ohne Einfluss auf die Schleifenbedingung

```
while( !a )
{
  Sleep( 1 ); // 1 Millisekunde
  ticks++;
}
```

Der Quellcode zeigt eine Endlosschleife. Der Compiler könnte hier einen Maschinencode erzeugen, der die Überprüfung der Variablen `a` nur ein einziges Mal durchführt, da er davon ausgeht, dass `a` innerhalb der Schleife sowieso nicht mehr verändert wird. Er könnte also den Code wie folgt umformen:

[49] Es gibt sogar Betriebssysteme, wie zum Beispiel QNX, die für konstante und variable Speicherinhalte unterschiedliche Segmente definieren.

Listing 2.97. Mögliches Ergebnis einer Compileroptimierung

```
if( !a )
{
  sprungmarke:
  Sleep( 1 ); // 1 Millisekunde
  ticks++;
  goto sprungmarke;
}
```

Die durch den Compiler umgeformte Variante hat den gleichen Ablauf wie die ursprüngliche. Es werden dabei nur wenige verschiedene Operationen ausgeführt, womit der Optimierer sein Ziel erreicht hat. Wenn die Variable `a` aber von einem separaten Ablaufstrang geändert werden kann, dann haben die beiden Varianten einen unterschiedlichen Verlauf. Der ursprüngliche Code kann zur Laufzeit beendet werden, der optimierte nicht.

Indem man die Variable `a` mit dem Schlüsselwort `volatile` deklariert, verbietet man dem Compiler solche Optimierungen im Zusammenhang mit dem Variablenzugriff. Er muss also in jedem Schleifendurchlauf die Variable testen.

```
volatile int a = 0;
```

Eine Variable mit der Speicherklasse `volatile` kann nicht gleichzeitig die Speicherklasse `register` erfüllen.

2.2.25 Der Scope - Operator

Der Scope-Operator `::` wurde schon in vielen vorangegangenen Abschnitten verwendet ohne ihn exakt zu erklären. Hier sollen nun alle Fälle durchgesprochen werden, die durch den Operator unterstützt werden. Wie schon gezeigt wurde, definiert er die Zugehörigkeit einer Definition zu einer Klasse, wenn diese außerhalb des eigentlichen Klassenrumpfes definiert wird. Das folgende Beispiel zeigt eine Methode `Umfang()`, die zu einer Klasse `Kreis` gehört:

Listing 2.98. Zuordnung einer Methode zu einer Klasse

```
double Kreis::Umfang() const
{
  // ...
}
```

Ebenso wie die Zugehörigkeit kann die Nichtzugehörigkeit ausgedrückt werden. Ein globale Funktion kann dadurch qualifiziert werden, dass man ihr den Scope-Operator voranstellt:

Listing 2.99. Globale Funktion

```
void ::f()
{
  // ...
}
```

Das gilt auch für den Aufruf von globalen Funktionen. Also `::f();`. Da es in C++ das argumentenabhängige Lookup-Verfahren gibt[50], kann eine solche Qualifizierung notwendig werden. Manche Softwareprojekte definieren es in ihren obligatorischen Coding Styles, dass globale Funktionen immer mit dem Scope-Operator qualifiziert werden müssen[51]. Dabei kann der Scopeoperator auch lokale von globalen Objekten unterscheiden:

Listing 2.100. Anwendung des Scope-Operators beim Ansprechen von Bezeichnern

```
int x;

int main()
{
  int x;

  x = 5;   // lokales x
  ::x = 7; // globales x

// ...
}
```

Von höherer Wichtigkeit wird der Scope-Operator beim Umgang mit Klassenhierarchien, wenn Elemente bestimmter Generationen ausgewählt werden sollen. Man nennt die exakte Beschreibung eines Elements durch den Scope-Operator *Qualifizierung*.

Listing 2.101. Qualifizierung durch den Scope-Operator

```
class A
{
public:
  void f()
  {
  // ...
  }
};
```

[50] Siehe Abschnitt 6.1.3 auf Seite 361.

[51] Davon müssen mit `#define` definierte Funktionsmakros ausgenommen werden, denn der Präprozessorlauf kennt nur die Syntax der Präprozessoranweisungen. Für ihn sind Scopes nicht existent.

```
class B : public A
{
public:
  void f() // überschreibt f() aus A
  {
    A::f(); // ruft f() aus A
    // ...
  }
};

int main()
{
  B b;

  b.f();    // ruft f() aus B
  b.A::f(); // ruft f() aus A

  // ...
}
```

Bei der Verwendung von Mehrfachvererbung kann der Scope-Operator ähnliche Dienste erweisen.

Listing 2.102. Qualifizierung bei Mehrfachvererbung

```
class X
{
  public:
    int x;
};

class A1 : public X
{};

class A2 : public X
{};

class B : public A1, public A2
{};

int main()
{
  B b;
  b.x = 0;     // Compilefehler!
  b.X::x = 0;  // Compilefehler!
```

```
  b.A1::x = 0; // OK
  b.A2::x = 0; // OK

 // ...
}
```

Die Klasse `B` enthält zwei `x`, da sie über zwei Richtungen die Basisklasse `X` bekommen hat. Ein unqualifizierter Zugriff auf `x` ist also zweideutig. Die Qualifizierung mit `X::x` bleibt zweideutig, da in `B` ja zwei Anteile der Klasse `X` enthalten sind. Durch die Vererbung sind die beiden Elemente `x` auch Elemente von `A1` bzw. von `A2`. Und genau darüber lassen sie sich unterscheiden.

Der Scope-Operator wird auch dazu verwendet, die Zugehörigkeit zu einem Namensraum zu kennzeichnen. Das wird syntaktisch ebenso ausgedrückt wie die Zugehörigkeit irgendwelcher Elemente zu ihrer Klasse. Dazu mehr im Abschnitt 6.1.2 auf Seite 353. Interessant wird der Scope-Operator, wenn das argumentenabhängige Lookup-Verfahren umgangen werden soll:

Listing 2.103. ADL versus Qualifizierung

```
namespace A
{
  class X {};

  void f(const X&)
  {
  }
}

namespace B
{
  void f(const A::X&)
  {
  }
}

int main()
{
  A::X x;

  f(x);   // Compiler findet A::f() durch ADL
  A::f(x) // Compiler findet A::f() durch Qualifizierung
  B::f(x) // ruft B::f()

  // ...
}
```

Wie das argumentenabhängige Lookup-Verfahren – Argument Dependent Lookup ADL – genau funktioniert, wird im Abschnitt 6.1.3 auf Seite 361 erklärt. An dieser Stelle soll einfach hingenommen werden, dass dieses Lookup-Verfahren des Compilers dazu führt, dass im Listing die Funktion `f()` aus dem Namensraum `A` dadurch gefunden wird, dass der Typ ihres Parameters `x` aus demselben Namensraum stammt.

2.2.26 Verschachtelte Typen

Klassen können auch ineinander verschachtelt werden. Man macht das insbesondere dann, wenn eine Klasse nur im Kontext einer anderen Klasse verwendet werden soll. Die eingeschachtelte Klasse ergibt möglicherweise ohne die umgebende Klasse überhaupt keinen Sinn. Also zum Beispiel:

Listing 2.104. Verschachtelte Klasse

```
class Toyota
{
public:
  // ...

  class ToyotaWagenheber
  {
    // ...
  };
};
```

Eine solche innere Klasse ist ohne ein Objekt der äußeren zwar instanziierbar, ergibt aber nur im Zusammenhang mit einem solchen Objekt wirklich Sinn. Bei der Instanziierung muss mit dem Scope-Operator der Name der inneren Klasse qualifiziert werden.

Listing 2.105. Instanziierung einer verschachtelten Klasse

```
// ...
Toyota::ToyotaWagenheber objekt;
// ...
```

Man kann eine innere Klasse auch verstecken, wenn sie nur innerhalb der Implementierung der äußeren Klasse gebraucht wird.

Listing 2.106. Private verschachtelte Klasse

```
class Citroen
{
  // ...
```

```
private:
  class CitroenLuftfederung
  {
    // ...
  };
};
```

Von außen ist diese innere Klasse nicht mehr zugreifbar. Derjenige, der die Klasse `Citroen` verwendet, hat zwar etwas von der Funktionalität der Klasse `CitroenLuftfederung`, kann diese Klasse aber nicht direkt verwenden. Sie wird innerhalb der äußeren Klasse benutzt.

Mit einer solchen Verschachtelung wird etwas modelliert, was ohne eine solche auch möglich wäre. Man bringt damit nur Zusammengehörigkeitsbeziehungen zum Ausdruck, die man anders nicht sehen würde.

2.2.27 Die `friend`-Deklaration

Das Schlüsselwort `friend` erlaubt ein Aufbrechen des Kapselungsprinzips. Eine Klasse kann Funktionen, Klassen und Klassenmethoden eine `friend`-Stellung einräumen. Das bedeutet, dass die entsprechende fremde Klasse oder die Funktion Zugriff auf alle privaten und geschützten Elemente der entsprechenden Klasse hat. Natürlich sollte eine solche Beziehung zwischen Klassen nur eingeführt werden, wenn dafür ein guter Grund besteht, schließlich bricht sie mit dem elementaren Kapselungsprinzip. In gewisser Hinsicht ist die `friend`-Stellung eine Ausnahme, die in Ausnahmefällen eingesetzt werden sollte[52].

Die Syntax in C++ sieht folgendermaßen aus:

Listing 2.107. Eine `friend`-Klasse

```
class A
{
public:
// ...
private:
  int x;
  friend class B;
};

class B
{
```

[52] Ein Beispiel für eine solche Ausnahme wird mit dem Barton-Nackman-Trick im Abschnitt 4.1.11 ab Seite 242 beschrieben, ein weiteres als kleines Detail im Abschnitt 3.3 im Listing auf Seite 205.

```
public:
  void f();
};

void B::f()
{
  A a;
  a.x = 5; // Zugriff nur durch friend-Stellung erlaubt.
}
```

Eine Funktion wird folgendermaßen zu einer `friend`-Funktion:

Listing 2.108. Eine `friend`-Funktion

```
void global_f();

class A
{
  // ...

  friend void global_f();
};
```

Und zuletzt eine Klassenmethode:

Listing 2.109. Eine `friend`-Methode

```
class A
{
  // ...

  friend void B::f();
};
```

Dabei ist es ganz egal, in welchem Sichtbarkeitsbereich eine solche `friend`-Deklaration steht. Ob `public`, `protected` oder `private`, es hat auf die modellierte Beziehung keinen Einfluss.

2.2.28 Statische Methoden und Attribute

Das Schlüsselwort `static` hat im Zusammenhang mit Klassenattributen eine ganz ähnliche Bedeutung wie bei der Deklaration statischer Daten innerhalb einer Funktion. Es wird die Lebensdauer eines Objekts beeinflusst, das als Attribut einer Klasse deklariert ist. Statische Daten haben eine Lebensdauer über die ganze Programmlaufzeit hinweg und sind unabhängig von Objekten der Klasse, in der sie deklariert wurden. Solche Elemente existieren zur

Laufzeit also unabhängig davon, ob irgendein Objekt der sie umfassenden Klasse instanziiert wurde. Sie sind das Gegenkonzept zur Instanzvariablen. Die Instanzvariable ist ein Attribut, das durch die Instanziierung eines Objektes zum Leben erweckt wird. Statische Attribute sind im Grunde genommen versteckte globale Daten. Das ist einer der Gründe, warum ihr Einsatz sehr sorgfältig abgewogen werden sollte. Instanzvariablen existieren zahlenmäßig so viele, wie eben Objekte einer Klasse instanziiert werden. Statische Daten sind dagegen singulär.

Ebenso objektunabhängig sind statische Methoden von Klassen. Diese Methoden können mit Hilfe des Scope-Operators direkt an dem Klassennamen aufgerufen werden. Da sie objektunabhängig sind, also auch keinen Zugriffsmechanismus auf ein Objekt besitzen, können sie nicht auf Instanzvariablen oder Instanzmethoden zugreifen, nur auf andere statische Methoden und auf statische Attribute. Statische Methoden sind versteckte globale Methoden. Die Zugehörigkeit zu einer Klasse räumt sie gewissermaßen etwas auf und entfernt sie aus dem globalen Scope.

Listing 2.110. Instanzzählung mit statischen Elementen

```
class X
{
public:
  X() { neueInstanz(); }
  X( const X & ) { neueInstanz(); }
  static unsigned long anzahlInstanzen();
private:
  static void neueInstanz();
  static unsigned long instanzen;
};

// ...

unsigned long X::instanzen = 0;

void X::neueInstanz()
{
  instanzen++;
}

unsigned long X::anzahlInstanzen()
{
  return instanzen;
}
```

Wie schon gesagt, sollten statische Elemente von Klassen nur dann verwendet werden, wenn es einen guten Grund dafür gibt. Das ist eher der Ausnahmefall als die Regel. Eine typische Anwendung ist die Referenzzählung wie in dem gezeigten Beispiel. Eine andere Anwendung steht im Zusammenhang mit der Serialisierung von Objekten. Das Schreiben von Objekten in einen Stream kann von Instanzmethoden übernommen werden. Der Interpretationsvorgang beim Einlesen kann nicht durch Instanzmethoden bewerkstelligt werden, da ein Objekt der entsprechenden Klasse noch nicht existiert. In diesem Kontext werden häufig statische Klassenmethoden eingesetzt.

2.2.29 Vererbung

Die Vererbung, auch Ableitung genannt, ist neben der Klassenbildung ein zentrales Konzept der Objektorientierten Programmierung. In C++ können Klassen und auch Strukturen vererbt werden[53]. Die erbende Klasse bekommt damit alle Elemente der Basisklasse. Allerdings wird die Vererbung nicht für eine Kumulierung von Elementen genutzt[54]. Vor allem sollte man in der Vererbung kein Konzept zur Wiederverwendbarkeit sehen. Es ist ein Instrument zur Flexibilisierung, indem es ermöglicht, sprachliche Abstraktion explizit im Entwurf von Software zu nutzen. Basisklassen werden also gebildet, um Abstraktionen darzustellen. Zur Abstraktion nutzen wir in den natürlichen Sprachen die Überbegriffsbildung. Diese abstrakten Überbegriffe fassen konkretere Begriffe zusammen. Überbegriffe bilden wir anhand bestimmter Merkmale der konkreten Begriffe, die wir mit ihnen zusammenfassen. Beispielsweise ist ein *Fahrzeug* etwas, das sich fortbewegen kann. Es ist ein sehr abstrakter Begriff, der sehr unterschiedliche konkrete Dinge zusammenfassen kann. So ist beispielsweise ein *Schiff* ebenso ein Fahrzeug wie auch das *Auto*. Die gemeinsame Eigenschaft manifestiert sich darin, dass sie sich mit einer gewissen Geschwindigkeit bewegen können. Diese ist zwar für jedes Fahrzeug verschieden, kann aber als Grundeigenschaft eines Fahrzeugs bezeichnet werden.

Listing 2.111. Vererbung

```
class Fahrzeug
{
public:
  int geschwindigkeit;
};
```

[53] Strukturen werden in C++ generell wie Klassen behandelt.

[54] Diese Denkweise herrschte noch Ende der 80er und Anfang der 90er Jahre bei der Mehrheit der Programmierer vor, die OO-Programmierung betrieben.

```
class Auto : public Fahrzeug
{
};

class Schiff : public Fahrzeug
{
};
```

In diesem Beispiel erben die Klassen `Auto` und `Schiff` von der Basisklasse Fahrzeug. Damit haben sie beide das Attribut `geschwindigkeit` aus der Basisklasse bekommen. Sie haben aber noch etwas viel Wichtigeres bekommen. Mit einer solchen Vererbung hat man erklärt, dass ein `Schiff` ein `Fahrzeug` ist. Ebenso für das `Auto`. Das ist ein sehr wichtiger Gedanke und wir werden später noch genauer darauf eingehen, was es für Konsequenzen hat, eine solche Erklärung abzugeben. An dieser Stelle soll sehr deutlich darauf hingewiesen werden, dass Vererbungen zur Modellierung von Überbegriffsbildungen[55] verwendet werden. Als Messinstrument für gelungene Vererbungsbeziehungen kann also auch zuallererst die Sprache herangezogen werden. Unser Sprachgefühl muss die Logik einer Überbegriffsbildung absegnen können. Es ist also einfach eine bestimmte sprachliche Logik, die hinter der Vererbung in der Objektorientierung steht[56]. Eine syntaktische Regel, die C++ mit allen anderen mir bekannten objektorientierten Sprachen teilt, ist, dass mit Referenzen oder Zeigern des Basisklassentyps auf Objekte der abgeleiteten Klassen verwiesen werden kann.

Listing 2.112. Verweis eines Kindklassenobjekts mit einem Basisklassenzeiger

```
...
int main()
{
  Fahrzeug *p = new Schiff();

  ...
```

[55] In der Linguistik auch Hyperonymbeziehung genannt.

[56] Der Vererbungsbegriff stammt natürlich aus der Biologie. Trotzdem sollte man nicht allzu lange über die biologische Vererbung meditieren, um das Wesen der Klassenvererbung in der Objektorientierten Programmierung zu verstehen. Die Biologie wurde im 20sten Jahrhundert eine sehr erfolgreiche Wissenschaft. Mancher spricht auch davon, dass die Biologie die Physik als „Leitwissenschaft“ abgelöst hätte. Das heißt, dass sie seit dem 20sten Jahrhundert prototypische Denkstrukturen auch für ganz andere Bereiche zur Verfügung stellt. Aufgrund ihres Erfolges wurden diese Denkweisen auch gerne von dieser neuen strahlenden Wissenschaft in ganz andere Zusammenhänge übernommen. Manchmal waren es auch weniger die kompletten Gedanken, als vielmehr die Begriffe. Der Begriff der Vererbung ist in der Objektorientierung also nur dem Wort nach mit dem in der Biologie identisch. Das, was das Wort „Vererbung“ bezeichnet, ist in diesen Bereichen etwas völlig Unterschiedliches.

```
  delete p;

  return 0;
}
```

So, wie man in einer natürlichen Sprache mit dem Begriff „Fahrzeug“ auf ein Schiff verweisen kann, kann man nun ebenso mit Zeigern und Referenzen des Typs `Fahrzeug` auf ein konkretes Objekt des Typs `Schiff` verweisen. Da dieser Gedanke erst im Laufe der Entwicklung von C++ in der nötigen Schärfe zu Tage trat, gibt es einige Merkmale in der Sprache, die aus älteren Ideen heraus geboren wurden – Merkmale, die heute wahrscheinlich nicht mehr so definiert werden würden. Dazu gehört beispielsweise, dass eine Vererbung standardmäßig eine private Sichtberkeit hat, wenn man sie nicht anders definiert. In dem Beispiel mit den Fahrzeugen steht hinter dem Doppelpunkt, der die Vererbung einleitet, das Schlüsselwort `public`. Stünde dort nichts, wäre `Fahrzeug` als Basisklasse für `Auto` und `Schiff` nach außen nicht sichtbar. Man könnte also nicht mit einem Zeiger oder einer Referenz des Basistyps auf Objekte der abgeleiteten Klasse verweisen. Das ergibt nach der heutigen Auffassung von Vererbung in der Objektorientierten Programmierung keinen Sinn. Es würde einen Entwurf unnötigerweise verkomplizieren. Ebenso verhält es sich mit der geschützten Vererbung, die zwar manchmal genutzt wird, um bestimmte, sehr C++-spezifische Tricks zu verwirklichen, aber eigentlich keinen Sinn für die OO-Programmierung hat. Solcherlei Tricks haben auch eher die Tendenz, den Code schlechter lesbar zu machen, und es existiert meistens eine bessere Lösung für das entsprechende Problem.

Vererbung und Sichtbarkeit

Syntaktisch hat also die Vererbung eine Sichtbarkeit. Standardmäßig ist die Vererbung eine private Sichtbarkeit. Diese kann aber geöffnet werden, so dass eine geschützte oder eine öffentliche Vererbung entsteht. Sinnvoll für die objektorientierte Entwicklung ist davon nur die öffentliche Vererbung, da es für die beiden anderen keine Deutungsmuster gibt. Die öffentliche Vererbung modelliert die Hyperonymbeziehung der natürlichen Sprachen. Wie diese Beziehung für die Softwareentwicklung genutzt werden kann, wird im Abschnitt 2.2.30 auf Seite 121 genauer erklärt. Voraussetzung für das Verständnis der Sinnhaftigkeit der Überbegriffsbildung ist jedoch, dass man die Vererbungsregeln kennt.

Listing 2.113. Sichtbarkeit in der Vererbung

```
class Basis
{
// ...
};
```

```
class S1 : Basis
{
// ...
};

class S2 : public Basis
{
// ...
};

class S3 : protected Basis
{
// ...
};

class S4 : private Basis
{
// ...
};
```

Im obigen Listing stehen fünf Klassen: die Basisklasse `Basis` und die davon abgeleiteten Klassen `S1`, `S2`, `S3` und `S4`. Bei der Ableitung von `Basis` auf `S1` ist kein Sichtbarkeitsbezeichner angegeben. Daher entspricht die Ableitung von `S1` der von `S4`, denn die Standardsichtbarkeit der Vererbung ist privat. Die Klasse `S3` hat einen geschützten Basisklassenanteil von `Basis`. Das heißt, dass Elemente von `Basis` nur innerhalb von `S3` und allen weiter abgeleiteten Klassen zugreifbar sind. Die private Vererbung von `Basis` nach `S1` und `S4` bewirkt, dass die Elemente Basisklasse nur innerhalb der direkt abgeleiteten Klassen zugegriffen werden können. Werden diese Klassen weiter abgeleitet, sind in den folgenden Generationen die Elemente von `Basis` nicht mehr zugreifbar.

Die Elemente der Basisklasse können nun innerhalb ihrer eigenen Klasse auch Sichtbarkeiten besitzen. Was das für die Sichtbarkeit in einer abgeleiteten Klasse bedeutet, soll hier kurz diskutiert werden:

Listing 2.114. Elementsichtbarkeit bei standardmäßig privater Vererbung

```
class Basis
{
public:
  int b1;
protected:
  int b2;
private:
  int b3;
};
```

```
class S1 : Basis
{
public:
  void f();
};

void S1::f()
{
  b1 = 0; // OK
  b2 = 0; // OK
  b3 = 0; // Nicht erlaubt! Compilefehler.
}

int main()
{
  S1 obj;
  obj.b1 = 0; // Nicht erlaubt! Compilefehler.
  obj.b2 = 0; // Nicht erlaubt! Compilefehler.
  obj.b3 = 0; // Nicht erlaubt! Compilefehler.

  return 0;
}
```

Die Klasse `S1` erbt privat von `Basis`. Das hat zur Folge, dass Elemente von `Basis` in `S1` nur innerhalb des Gültigkeitsbereichs von `S1` zugreifbar sind. Der Basisklassenanteil kann also nur innerhalb von `S1` beeinflusst werden. Das zeigt die Methode `S1::f()`. Allerdings gelten auch dort die Sichtbarkeitsregeln, die in `Basis` schon festgelegt wurden. So ist das Datenelement `b3` in `Basis` privat deklariert worden und deshalb nirgends außerhalb von `Basis`, also auch nicht in der abgeleiteten Klasse `S1` zugreifbar.

Listing 2.115. Elementsichtbarkeit bei öffentlicher Vererbung

```
class Basis
{
public:
  int b1;
protected:
  int b2;
private:
  int b3;
};

class S2 : public Basis
{
```

```
public:
  void f();
};

void S2::f()
{
  b1 = 0; // OK
  b2 = 0; // OK
  b3 = 0; // Nicht erlaubt! Compilefehler.
}

int main()
{
  S2 obj;
  obj.b1 = 0; // OK
  obj.b2 = 0; // Nicht erlaubt! Compilefehler.
  obj.b3 = 0; // Nicht erlaubt! Compilefehler.

  return 0;
}
```

Bei der öffentlichen Vererbung kann auf Basisklassenanteile in einer abgeleiteten Klasse auch von außerhalb zugegriffen werden. Natürlich vererben sich auch dort die Sichtbarkeiten der Elemente, wie sie innerhalb der Basisklasse definiert wurden. Private Elemente der Basisklasse bleiben also in ihrer Zugriffsmöglichkeit auf die Basisklasse beschränkt.

Listing 2.116. Elementsichtbarkeit bei geschützter Vererbung

```
class Basis
{
public:
  int b1;
protected:
  int b2;
private:
  int b3;
};

class S3 : protected Basis
{
public:
  void f();
};
```

```
void S3::f()
{
  b1 = 0; // OK
  b2 = 0; // OK
  b3 = 0; // Nicht erlaubt! Compilefehler.
}

int main()
{
  S3 obj;
  obj.b1 = 0; // Nicht erlaubt! Compilefehler.
  obj.b2 = 0; // Nicht erlaubt! Compilefehler.
  obj.b3 = 0; // Nicht erlaubt! Compilefehler.

  return 0;
}
```

Die geschützte Sichtbarkeit sieht in der ersten Generation aus wie eine private. Innerhalb der abgeleiteten Klasse sind die Elemente zugreifbar, wenn sie nicht schon in der Basisklasse privat deklariert wurden. Der Unterschied zur privaten Vererbung zeigt sich erst bei einer weiteren Vererbung von `S3`. Dann sind die Elemente aus `Basis` weiter zugreifbar.

Die Sichtbarkeit von Klassenelementen wird durch die Sichtbarkeit der Vererbung eingeschränkt, wenn diese strenger ist. Beispielsweise wird aus einem öffentlichen Element einer Basisklasse ein geschütztes, wenn die Vererbung geschützt ist, usw.

Es gibt auch die Möglichkeit, die Sichtbarkeit im Nachhinein zu öffnen. Dazu dient die `using`-Direktive innerhalb von Klassen.

Listing 2.117. Elementsichtbarkeit bei privater Vererbung

```
class Basis
{
public:
  int b1;
protected:
  int b2;
private:
  int b3;
};

class S1 : private Basis
{
public:
  void f();
```

```
  using Basis::b1;
  using Basis::b2;
};

void S1::f()
{
  b1 = 0; // OK
  b2 = 0; // OK
  b3 = 0; // Nicht erlaubt! Compilefehler.
}

int main()
{
  S1 obj;
  obj.b1 = 0; // OK
  obj.b2 = 0; // OK
  obj.b3 = 0; // Nicht erlaubt! Compilefehler.

  return 0;
}
```

Die private Vererbung führt zunächst dazu, dass auf die öffentlichen und geschützten Elemente der Klasse `Basis` nur innerhalb von `S1` zugegriffen werden kann. Diese Elemente wurden innerhalb von `S1` zunächst auf private Sichtbarkeit eingeschränkt. Durch die `using`-Direktive aber kann man ihnen eine neue Sichtbarkeit verleihen, indem man die Basisklassenelemente damit in dem Block der gewünschten Sichtbarkeit aufführt.

Auch diese Möglichkeit, Vererbungsbeziehungen zu modifizieren sollte äußerst sparsam eingesetzt werden, denn es fehlt auch hier die nötige Interpretierbarkeit durch den Menschen selbst. Diese Mittel haben also für die eigentliche objektorientierte Entwicklung keine Bedeutung und stehen einem sauberen Entwurf eher entgegen. Da C++ aber nicht nur einem Paradigma folgt, können die Regeln im Zusammenhang mit der Sichtbarkeit bei der Vererbung an ganz anderer Stelle wie zum Beispiel der Templateprogrammierung wieder wichtig werden.

2.2.30 Virtuelle Methoden und Polymorphismus

In diesem Abschnitt soll die Ernte dafür eingefahren werden, dass wir uns im vorangegangenen Abschnitt mit den trockenen Vererbungsregeln von C++ befasst haben. Welche Bedeutung hat also die Überbegriffsbildung für den Entwurf von Software?

Überbegriffe fassen Dinge zusammen, die unterschiedliche Eigenschaften und unterschiedliches Verhalten besitzen. In der Objektorientierung wird Verhalten durch Methoden definiert. Die Basisklassen, die die Überbegriffe darstellen, enthalten üblicherweise Methodendeklarationen, aber keine Implementierungen der Methoden.

In der Verwendung von Basisklassen wird ein sprachlicher Zusammenhang zur technischen Konstruktion genutzt. Die Vererbung in der Objektorientierung bildet also nichts ab, was man analog zur biologischen Vererbung sehen könnte. Die Vererbung in der OO-Programmierung bildet die sprachliche Überbegriffsbildung ab. Die Testfrage für die Überprüfung der Gültigkeit und der Sinnhaftigkeit von Vererbung lautet deshalb: „Ist X ein Y?". Also beispielsweise: „Ist das Pferd ein Säugetier?". Der Begriff des Säugetiers nimmt hier die Überbegriffsstellung zum Begriff „Pferd" ein, ebenso wie auch die Begriffe „Löwe" oder „Maus" damit abgedeckt sind.

Dabei implementiert das Pferd die Eigenschaft, die im Begriff Säugetier beschrieben wird. Diese Eigenschaft findet sich bekanntlich sehr unterschiedlich realisiert in den vielen unterschiedlichen Arten der Säugetiere. Dabei gibt es keine „reine" Verwirklichung der Säugetiereigenschaft im Begriff Säugetier selbst[57]. Dieser Begriff ist abstrakt. Er verspricht etwas, was nur bei konkreten Tierarten anzutreffen ist.

Von diesem Gedanken aus versuchen wir nun in die Objektorientierte Programmierung zu kommen und die nötigen Regeln von C++ zu betrachten. Nehmen wir an, es soll ein Programm geschrieben werden, das Vektorgrafiken darstellen soll. Innerhalb der Vektorgrafiken gibt es Einzelelemente – die Bestandteile der Grafiken sind Punkte, Linien, Rechtecke, Kreise, Dreiecke und Polygone. Ebenso wie wir diese Elemente mit sprachlichen Begriffen zusammenfassen um das Beschriebene zu vereinfachen (in diesem und dem vorangegangenen Satz wurden die Begriffe „Einzelelemente", „Bestandteile" und „Elemente" dazu verwendet), kann in der Programmierung ein solcher abstrakter Standpunkt gefunden werden, der die Implementierung vereinfacht, indem er an anderer Stelle in den Funktionen die Unterscheidung nach verschiedenen Sachverhalten überflüssig macht. In dem Gedankenexperiment sollen die Elemente der Vektorgrafik „Grafikelemente" genannt werden. Damit wird die Modellierung einer Klasse mit diesem Namen notwendig.

Listing 2.118. Einfache Methode

```
class Grafikelement
{
public:
  void zeichne( Zeichenflaeche *zf );
};
```

[57] Hat schon jemand das reine Säugetier gesehen?

Grafikelemente zeichnen sich dadurch aus, dass sie auf irgendeinen Hintergrund gezeichnet werden können, beispielsweise auf einen Fensterhintergrund oder auf Papier. Der Begriff „Zeichenfläche“ ist für die angesprochenen konkreten Ausgabeflächen ein sprachlicher Überbegriff und abstrahiert diese damit. Diese Abstraktion kann man sich ebenso zunutze machen wie die des Grafikelements. Sie soll aber an dieser Stelle nicht weiterverfolgt werden, um die Sache nicht zu erschweren[58]. Wichtig ist, dass sich jedes Grafikelement in einer eigenen Weise zeichnet. Die Klasse `Grafikelement` kann keine eigene Implementierung dafür angeben. In einem Code wie diesem:

Listing 2.119. Methodenaufruf

```
// Pseudocode
void Ausgabefenster::zeichne( Grafikelement *e )
{
  e->zeichne( liefereZeichenflaeche() );
}
```

muss also zur Laufzeit die Implementierung der Methode `zeichne()` gefunden werden. Die Implementierung kann die der Klassen „Kreis“ oder „Polygon“ sein, je nachdem, welcher Objekttyp gerade mit dem Zeiger `e` referenziert wird. Die Eigenschaft einer Methode, dass sie zur Laufzeit gefunden werden kann, muss man mit dem Schlüsselwort `virtual` deklarieren. Also:

Listing 2.120. Virtuelle Methode

```
// Pseudocode
class Grafikelement
{
public:
  virtual void zeichne( Zeichenflaeche *zf );
};
```

Damit fügt der Compiler Code ein, der die so genannte *späte Bindung* ermöglicht. Es bleibt aber noch ein Problem: die Klasse `Grafikelement` kann die Methode `zeichne()` gar nicht implementieren, da in dieser Abstraktionsebene noch gar nicht klar ist, was implementiert werden soll. Diesem Problem kommt C++ mit einem besonderen syntaktischen Ausdruck entgegen.

Listing 2.121. Rein virtuelle Methode

```
// Pseudocode
class Grafikelement
{
```

[58] Bekannte und verbreitete Frameworks zur GUI-Programmierung verfolgen exakt diesen Ansatz. So gibt es in der schon betagten MFC einen „Device Context“, der einen solchen Zeichenhintergrund darstellt. In den Java-Frameworks zur Fensterprogrammierung heißt das entsprechende Element „Graphics“.

```
public:
  virtual void zeichne( Zeichenflaeche *zf ) = 0;
};
```

Mit der Kennzeichnung „= 0“ am Ende der Methodendeklaration wird ausgedrückt, dass die Methode für diese Klasse nicht deklariert werden kann[59]. Damit entfällt auch die Notwendigkeit die Methode zu definieren[60]. Man spricht von einer „rein virtuellen“ oder auch von einer „abstrakten“ Methode. Dabei gilt die Regel, dass eine Klasse, wenn sie mindestens eine rein virtuelle Methode besitzt, selbst abstrakt ist und nicht als Objekt instanziiert werden kann. Klassen mit rein virtuellen Methoden können also nicht zur Objektinstanziierung herangezogen werden. Das gilt auch für abgeleitete Klassen, die die entsprechenden Methoden nicht implementieren. Klassen werden dadurch konkret, dass sie alle rein virtuellen Methoden implementieren – wie die Klasse `Linie` im folgenden Listing.

Listing 2.122. Implementierung einer rein virtuellen Methode in einer Kindklasse

```
// Pseudocode
class Linie : public Grafikelement
{
public:
  Linie( const Punkt &a, const Punkt &e )
  : p1(a), p2(e) {}
  void zeichne( Zeichenflaeche *zf );
private:
  Punkt p1, p2;
};

void Linie::zeichne( Zeichenflaeche *zf )
{
  zf->positioniere( p1.x, p1.y );
  zf->zeichneLinie( p2.x, p2.y );
}
```

Das Verständnis der virtuellen Methoden ist für die Beherrschung der objektorientierten Programmierung unter C++ zentral. Was man in C++ „virtuelle“ Methode nennt, nennt man in anderen OO-Sprachen „spätgebundene“ Methode. Man benötigt beispielsweise in Java das Schlüsselwort `virtual` nicht, um dieses Prinzip zu nutzen. Jede öffentliche Objektmethode ist dort standardmäßig spätgebunden. Mit dieser späten Bindung realisiert

[59] Die Null hat in diesem Zusammenhang keine numerische Bedeutung.

[60] Da es für gewöhnlich keinen Sinn ergibt, eine virtuelle Methode für die Basisklasse zu definieren, brachte diese Kennzeichnungsmöglichkeit ab der AT&T-Version 2.0 einen enormen Vorteil. Vorher mussten diese Funktionen auch definiert werden.

man den dynamischen Polymorphismus[61], der eine technische Entsprechung der sprachlichen Überbegriffsbildung ist. Es handelt sich hierbei um den eigentlichen „Kern“ objektorientierter Programmierung.

Der virtuelle Destruktor

Beim Zerstören von Objekten in C++ muss der Destruktor aufgerufen werden. Manche Objekte müssen beispielsweise Speicherbereiche freigeben, Dateien schließen oder Netzverbindungen abbrechen. Dazu brauchen sie den Destruktoraufruf. Wenn Objekte polymorph sind, das heißt, wenn sie unterschiedlich sind und anhand des gleichen Basisklassenzeigertyps verwaltet werden, werden sie normalerweise auch über diesen Zeigertyp gelöscht. Damit auch über einen Basisklassenzeiger beim Löschen der richtige Destruktor aufgerufen wird, muss dieser in der Basisklasse virtuell deklariert werden. Wie jede andere Methode auch, kann dann der Destruktor zur Laufzeit gefunden werden.

Listing 2.123. Virtueller Destruktor

```
class Basis
{
public:
  virtual ~Basis();
};

class Spezialisierung : public Basis
{
public:
  ~Spezialisierung();
};

// Typischer leerer Destruktor einer Basisklasse.
Basis::~Basis() {}

Spezialisierung::~Spezialisierung()
{
 // Dieser Destruktor hat eventuell etwas zu tun.
}

int main()
{
  Basis *obj = new Spezialisierung();
```

[61] „Dynamisch“ deshalb, da dieser Polymorphismus zur Laufzeit aufgelöst wird. Mit dem statischen Polymorphismus spricht man nur die Möglichkeit zur Funktionsüberladung an, die zur Compilezeit gelöst wird.

```
  delete obj;

  return 0;
}
```

Wenn der Destruktor einer Basisklasse virtuell deklariert wurde, sind die Destruktoren aller von ihr abgeleiteten Klassen virtuell.

Wenn eine Klasse als Basisklasse entworfen wird, muss daran gedacht werden, dass eine abgeleitete Klasse eventuell einen Konstruktor braucht. Insofern ist es die Regel, dass Basisklassen virtuelle Destruktoren enthalten.

Da die Basisklasse selbst kein Verhalten definiert und es demzufolge auch nichts aufzuräumen gibt, wird der Basisklassendestruktor meistens leer bleiben. Wichtig ist er trotzdem, da ohne ihn der Destruktor der Kindklasse nie erreicht werden wird. Das Beispiel aus dem vorangegangenen Abschnitt mit der Klasse `Grafikelement` müsste also korrekt so aussehen:

Listing 2.124. Typische Elemente einer Basisklasse

```
// Pseudocode
class Grafikelement
{
public:
  virtual ~Grafikelement();
  virtual void zeichne( Zeichenflaeche *zf ) = 0;
};

// In der CPP-Datei
Grafikelement::~Grafikelement() {}
```

Es ist also ein typisches C++-Idiom, dass Basisklassen leere virtuelle Destruktoren besitzen. Dabei kann ein virtueller Destruktor rein virtuell deklariert werden. Eine Implementierung muss er aber trotzdem besitzen, denn Destruktoren werden verkettet. Ein Kindklassendestruktor wird immer den Basisklassendestruktor aufrufen wollen.

Listing 2.125. Rein virtueller Destruktor

```
// Pseudocode
class Grafikelement
{
public:
  virtual ~Grafikelement() = 0;
  virtual void zeichne( Zeichenflaeche *zf ) = 0;
};

// In der CPP-Datei ist trotz der rein virtuellen
// Deklaration eine Implementierung notwendig!
Grafikelement::~Grafikelement() {}
```

Es gibt nur wenige Fälle, in denen eine rein virtuelle Deklaration des Destruktors einen echten Vorteil bietet. Man kann dadurch die Klasse als abstrakt kennzeichnen, wenn sie keine andere virtuelle Methode besitzt.

Ein weiteres Thema ist die Unterbringung des leeren virtuellen Destruktors in einer Implementierungsdatei (.cpp). Wenn man die Leerimplementierung inline in der Klasse definieren würde, hat man unter Umständen das Problem, dass der Compiler bei jedem Inkludieren der Headerdatei in ein Modul einen Destruktor in die Objektdatei des Moduls schreibt. Virtuelle Methoden können nicht inline expandiert werden. Das gilt auch für Leerimplementierungen. Diese sollten immer in einer Implementierungsdatei untergebracht werden, die der Compiler nur einmal pro Compilevorgang durchläuft. Das angesprochene Problem wird in Abschnitt 6.1.4 ab Seite 363 auch für andere Konstrukte beschrieben.

Fazit

Als Quintessenz aus diesem Abschnitt sollte man sich auf jeden Fall merken, dass Basisklassen in fast jedem Fall einen virtuellen Destruktor brauchen, da andernfalls die Destruktoren von Kindklassen in dem typischen Anwendungsfall des dynamischen Polymorphismus nicht aufgerufen werden können. Die Implementierungen dieser virtuellen Elternklassendestruktoren sind meistens leer.

2.2.31 Operatoren der Typenkonvertierung

In einigen vorangegangenen Beispielen wurden die Operatoren der Typenkonvertierung schon genannt. In diesem Abschnitt sollen sie systematisch betrachtet und dargestellt werden.

C++ kennt ja noch die alte Variante der Typenkonvertierung aus C. Der Zieltyp wurde in Klammern vor den zu konvertierenden Wert gesetzt: `(Zieltyp)Wert`. In C++ wurde diese Konvertierung mit einer weiteren syntaktischen Form versehen, der Konstruktorschreibweise: `Zieltyp(Wert)`. Diese beiden Schreibweisen lösen exakt den gleichen Vorgang aus. Die Konstruktorschreibweise ist nur aus einem Grund vorzuziehen, da sie deutlicher zeigt, was mit einer Konvertierung geschieht: es wird ein temporäres unbenanntes neues Objekt des Zieltyps erzeugt.

Seit Mitte der 90er Jahre wurden vier neue Schreibweisen für Typenkonvertierungen mit Hilfe von vier speziellen Operatoren in C++ eingeführt. Diese Operatoren heißen:

```
static_cast<>()
const_cast<>()
dynamic_cast<>()
reinterpret_cast<>()
```

Um den Sinn und die Funktionsweise dieser Operatoren zu verstehen, betrachten wir einmal die Anwendung der alten Konvertierungsweise etwas näher. In welchen Situationen werden Typenkonvertierungen durchgeführt?

`static_cast<>()`

Da ist zum einen die Konvertierung zwischen den Standarddatentypen, die Zahlenwerte repräsentieren. Diese Konvertierungen sind in der Sprache definiert. Wann Werte verloren gehen, wann gekürzt wird oder wann ein Überlauf stattfindet, hängt von den entsprechenden Standardtypen ab und lässt sich aus deren Breite bzw. deren Fassungsvermögen erklären. Alle diese durch die interne Beschaffenheit der Typen definierten Konvertierungen lassen sich mit dem Operator `static_cast<>()` ausdrücken. Dazu kommen die Konvertierungen in Klassentypen, die durch einparametrige Konstruktoren definiert sind. Ist ein Konstruktor für den Typ `X` vorhanden, der einen Typ `Y` entgegennimmt, dann kann ein statischer Cast von `Y` nach `X` durchgeführt werden. Der statische Cast führt also wohldefinierte Konvertierungen durch.

Listing 2.126. Gültige Konvertierungen mit `static_cast<>()`

```
class X
{
public:
  X( int i ) {}
};

void f( const X &obj )
{
// ...
}

int main()
{
  double d = 55.9;
  int i = static_cast<int>(d);

  f( static_cast<X>(i) );

  return 0;
}
```

`reinterpret_cast<>()`

Wenn Konvertierungen direkt zwischen Strukturen angewendet werden, die nicht miteinander in einer Vererbungshierarchie stehen, so handelt es sich

um eine Reinterpretation des Inhalts. Ein obskurer Vorgang, der oft sehr fragwürdig ist und in nur sehr seltenen Fällen Bestandteil eines wartbaren Codes sein wird. Vor allem ist die Konvertierung zwischen unterschiedlichen Zeigertypen außerhalb einer Vererbungshierarchie ein Vorgang, der eher einem Hack als strukturiertem Vorgehen gleicht. Solche Konvertierungen ergeben keinen aus den Typenbeschreibungen ableitbaren Sinn.

Hingegen ist die Konvertierung von Zeigern von und nach `void *` ein allgemein angewendeter und gut interpretierbarer Vorgang. Eine der häufigsten Situationen ist ein Aufruf der Funktion `malloc()` um Speicher zu allokieren. Die Funktion liefert einen `void *` Zeiger zurück, der in einen typisierten Zeiger konvertiert werden muss, über den man auf den Speicher zugreifen kann.

Beide genannten Konvertierungssituationen werden mit dem Operator `reinterpret_cast<>()` durchgeführt.

Auch die als problematisch beschriebene Konvertierung kann in bestimmten Fällen einen Sinn ergeben. So kann es beispielsweise notwendig sein einen einfachen Zahlenwert in einen Zeiger zu konvertieren, wenn irgendein Gerät an einer festen Adresse angeschlossen ist. Das ist auch der Hauptanwendungsbereich des Operators `reinterpret_cast<>()` – immer wenn es um Daten geht, die von außerhalb des C++-Typensystems geliefert werden. Wenn es um das Aufprägen von Typeninformation auf eine Anzahl untypisierter Bytes geht, dann ist der reinterpretierende Cast das Mittel der Wahl. Das Auftreten dieses Casts im Quellcode sollte auf das absolut mögliche Minimum begrenzt werden.

`const_cast<>()`

Eine seltene Form der Konvertierung ist es, wenn die `const`-Maskierung eines Zeigers oder einer Referenz entfernt werden soll. Das kommt immer dann vor, wenn ein Fehler in der Modellierung einer Klasse vorliegt und eine Methode, die keine Änderungen am Objekt durchführt, nicht `const`-spezifiziert wurde.

Listing 2.127. Fehlende `const`-Spezifikation

```
class Vector
{
// ...
public:
  unsigned getSize() { return size; }
private:
  unsigned size;
// ...
};

void foo( const Vector *v )
{
```

```
  unsigned s = v->getSize(); // Fehler!
// ...
}
```

Eine Funktion (oder eine Methode) maskiert ihre Parameter für gewöhnlich mit `const`, wenn an den Parameterobjekten nichts verändert werden soll. Das versetzt die Funktion auch in die Lage mit konstanten Objekten parametrisiert zu werden. Ohne die `const`-Maskierung könnte die Funktion nur Parameter erhalten, die beschreibbar, also veränderlich wären. Mit der Maskierung kann die Funktion `void foo( const Vector *v )` nur lesend auf das Objekt `v` zugreifen. Das wird versucht mit dem Aufruf von `getSize()`. Diese Methode ändert tatsächlich nichts am Objekt. Wenn diese Eigenschaft aber nicht deklariert ist, dann kann der Compiler das auch nicht wissen und moniert einen Fehler. Korrekterweise müsste also in der Klasse `Vector` die Methode `getSize()` folgendermaßen deklariert werden: `unsigned getSize() const;`. Da jedoch genau diese Deklaration mit der `const`-Spezifikation häufig vergessen wird, sind die Anwender einer solchen fehlerhaften Klasse manchmal zur Konvertierung des Objektzeigers oder der Objektreferenz gezwungen.

Listing 2.128. Korrekturcast bei fehlender `const`-Spezifikation

```
// ...
void foo( const Vector *v )
{
  Vector *p = const_cast<Vector*>(v);
  unsigned s = p->getSize();
// ...
}
```

Das ist zwar ein sehr problematischer Vorgang und sollte zur Nachbearbeitung der verantwortlichen Klasse führen. Ganz vermeiden lässt sich dieser Fall aber nicht. Aus diesem Grund ist es auch sinnvoll, dass man die problematischen Stellen im Code deutlich durch die Verwendung des Operators `const_cast<>()` sieht. Würde die alte Konvertierungsschreibweise verwendet werden, dann würde die entsprechende Stelle unauffällig in der Masse des Codes untergehen und schwer zu finden sein[62].

`dynamic_cast<>()`

Die einzige Konvertierung, die neu hinzugekommen ist und die nicht in der entsprechenden Form mit alten syntaktischen Mitteln emuliert wer-

[62] Man bedenke, dass die Anwendung des Schlüsselworts `const` nicht einfach nur kosmetischer Natur ist. Es wird vom Compiler zur Unterscheidung der Methodensignaturen herangezogen. Ein Objekt kann beispielsweise in einem nicht veränderbaren Speicherbereich liegen. Manche Betriebssysteme legen konstante Objekte in einem dafür ausgewiesenen Speicherbereich an und überprüfen die Zugriffe. Ein Schreibzugriff auf ein solches Objekt würde das Beenden des Prozesses bedeuten.

den kann ist die dynamische Konvertierung von Zeigern innerhalb einer Vererbungshierarchie. Der Operator, der dafür zur Verfügung steht, heißt `dynamic_cast<>()`. Während eine Konvertierung eines Kindklassenzeigers in einen Elternklassenzeiger implizit durchgeführt werden kann, ist die andere Richtung nur explizit möglich. Das hat seinen Grund darin, dass nicht garantiert werden kann, ob das Objekt, auf das der Zeiger zeigt, tatsächlich von dem genannten Zieltyp ist.

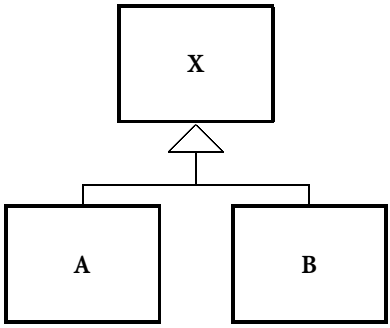


Abbildung 2.3. Einfache Vererbung

Wie in der Abbildung deutlich wird, kann jede Elternklasse mehrere Kindklassen haben. Wenn also ein Zeiger eines Elternklassentyps – hier `X` – in einen Kindklassentyp – z. B. `A` – konvertiert werden soll, muss der Fall ausgeschlossen werden, dass der Zeiger in Wirklichkeit auf ein Objekt einer anderen Kindklasse – evtl. `B` – zeigt. Der Operator `dynamic_cast<>()` führt diese Überprüfung durch. Wenn die Zeigerkonvertierung nicht durchgeführt werden kann, weil der Zeiger auf einen falschen Objekttyp zeigt, liefert der Operator einen Nullzeiger zurück. Die Verwendung des Operators ist aber an die Bedingung geknüpft, dass die Klassentypen, die an der Konvertierung beteiligt sind, virtuelle Funktionstabellen besitzen. Der Operator sucht den Zieltyp über diese Tabellen. Kann er ihn darüber nicht finden, gibt er den Nullzeiger zurück. Wenn man mit Referenzen statt mit Zeigern arbeitet, wirft `dynamic_cast<>()` im Fehlerfall eine Ausnahme: die `bad_cast`-Exception.

Listing 2.129. Dynamischer Downcast

```
#include <iostream>

class X
{
public:
    virtual ~X() {}
};
```

```
class A : public X {};
class B : public X {};

void f( X* p )
{
  using namespace std;
  A* ap = dynamic_cast<A*>(p);
  if( ap )
    cout << "p zeigt auf ein Objekt von A" << endl;
  else
    cout << "p zeigt auf kein Objekt von A" << endl;
}

int main()
{
    A a;
    B b;

    f( &a );
    f( &b );

    return 0;
}
```

In dem Beispiel überprüft die Funktion `f()` den Zeiger des Typs `X*` darauf, ob er auf ein Objekt vom Typ `A` zeigt. Die Anwendung des Konvertierungsoperators `dynamic_cast<>()` dafür hat die Voraussetzung, dass die Klasse `X` eine virtuelle Funktionstabelle besitzt. Aus diesem Grund hat `X` einen virtuellen Destruktor. Es kann aber auch eine beliebige andere virtuelle Methode sein. Eine V-Tabelle wird immer dann in eine Klasse aufgenommen, wenn mindestens eine virtuelle Methode in ihr deklariert wurde.

Neben einem überprüften Downcast[63] kann mit dem `dynamic_cast<>()`-Operator auch ein Sidecast durchgeführt werden. Eine Konvertierung eines Elternklassenzeigers in einen anderen, unabhängigen Elterklassentyp. Eine solche Konvertierung ist in dem Fall erfolgreich, wenn das Objekt von beiden Elternklassen öffentlich erbt. Ein solcher Sidecast ist also nur bei Mehrfachvererbung (siehe Abschnitt 2.2.32) einsetzbar.

Die Abbildung zeigt eine Situation, in der eine erfolgreiche Konvertierung eines Zeigers zwischen zwei unabhängigen Klassentypen denkbar ist. Ein Zeiger vom Typ `X*` kann dann in den Typ `Y*` konvertiert werden, wenn das Objekt, auf das er zeigt, vom Typ `B` ist. Handelt es sich um ein Objekt der Klasse `A`, dann wird die Konvertierung nicht erfolgreich sein.

[63] Downcast: Konvertierung in die Richtung der Kindklasse. Die Konvertierung in Richtung der Elternklasse nennt man Upcast.

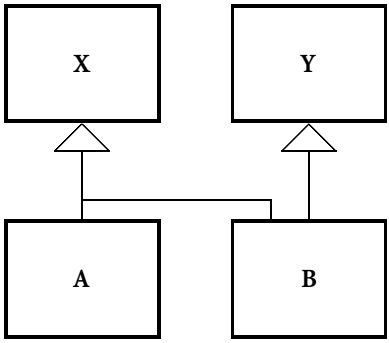


Abbildung 2.4. Mehrfachvererbung

Listing 2.130. Dynamischer Sidecast

```
#include <iostream>

class X
{
public:
    virtual ~X() {}
};

class Y
{
public:
    virtual ~Y() {}
};

class A : public X {};
class B : public X, public Y {};

void f( X* p )
{
  using namespace std;
  Y* yp = dynamic_cast<Y*>(p);
  if( yp )
    cout << "Konverierung nach Y* erfolgreich." << endl;
}

int main()
{
    A a;
    B b;
```

```
    f( &a );
    f( &b );

    return 0;
}
```

Zu beachten ist auch, dass `dynamic_cast<>()` natürlich nur auf solche Elternklassen konvertieren kann, die auch öffentlich sichtbar sind (sogenannte Schnittstellen)[64].

Mit den V-Tabellen nutzt der Konvertierungsoperator `dynamic_cast<>()` die Typeninformation, die im Compilat erhalten geblieben ist. Da es traditionell in C und C++ keine sprachlichen Strukturen gibt, die auf echte Typeninformation zur Laufzeit angewiesen sind, optimiert der Compiler alles heraus, was nicht unmittelbar zum Aufruf der virtuellen Funktionen benötigt wird. Um eine virtuelle Funktion zu finden, muss beispielsweise keine Rückbindung zur Laufzeit auf weitere Elternklassen erfolgen. Eine virtuelle Funktion wird immer in Vererbungsrichtung in der Kindklasse gesucht. Der Operator `dynamic_cast<>()` braucht aber diese Rückbindung, denn er muss die Elternklassen finden. Das erfordert eine Erweiterung der V-Tabellen um eben diese Information. Bei manchen Compilern, wie zum Beispiel dem Microsoft Visual C++-Compiler Versionen 6.0 und 7.x, muss diese Information extra aktiviert werden, damit der dynamische Cast funktioniert.

2.2.32 Mehrfachvererbung

Im Gegensatz zu der wesentlich neueren Sprache Java gibt es in C++ die Mehrfachvererbung von Klassen. Dass dieses Feature in Java nicht existiert, hat nichts damit zu tun, dass es eventuell für die Hersteller der Entwicklungswerkzeuge zu kompliziert wäre, es zur Verfügung zu stellen. Es hat vielmehr damit etwas zu tun, dass die Mehrfachvererbung ein enormes Potenzial besitzt, falsch angewendet zu werden[65]. Aus meiner Projekterfahrung kann ich berichten, dass von den Einsatzfällen der Mehrfachvererbung etwa 50% fehlerhaft sind und zu ernsthaften Folgeproblemen führen. Zur gewinnbringenden Anwendung der Mehrfachvererbung kann man eine Faustregel anwenden: *Man leite maximal von einer Klasse ab, die eine Implementierung enthält*[66]. Alle anderen Elternklassen sollten ausschließlich Deklarationen rein virtueller Methoden enthalten.

[64] Sogar ein Upcast eines Kindklassenzeigers auf den Elterklassentyp schlägt fehl, wenn die Vererbung privat oder geschützt ist.

[65] Java besitzt eine alternative Technik: die Mehrfachimplementierung von Interfaces. Diese ist viel besser gegen falsche Anwendung gewappnet als die Mehrfachvererbung von Klassen.

[66] Man nennt das Ableiten von einer Klasse, die Implementierung enthält, auch „Implementierungsvererbung".

Damit diese Aussage nicht nur eine Behauptung darstellt, soll sie theoretisch untermauert werden. Das Problem der Mehrfachvererbung besteht auf keinen Fall in einer möglichen Doppeldeutigkeit mancher Bezeichner, wie es manchmal in etwas oberflächlicher Literatur postuliert wird. Doppeldeutigkeiten lassen sich mit Qualifizierung durch den Scope-Operator mit dem entsprechenden Elternklassentyp auflösen. Das Problem ist vielmehr struktureller Natur. Einen Zugang zu dem Problem gibt vielleicht der folgende Gedanke:

Nehmen wir einmal an, dass in einem Ingenieurbüro ein Amphibienfahrzeug konstruiert werden soll – ein Automobil, dass sich auch im Wasser schwimmend fortbewegen kann. Dieses Fahrzeug hat etwas von einem herkömmlichen Auto, wie auch von einem Boot. Trotzdem wird es nicht möglich sein, ein schon vorhandenes Auto mit einem schon existierenden Boot so zu vereinen, dass daraus ein neues Fahrzeug entsteht. Ein solches Amphibienfahrzeug muss nur der Idee des Bootes genauso genügen wie der Idee des Automobils. In seinen Hauptmerkmalen wird man es neu konstruieren müssen. In seinen Einzelteilen wird man sicher auf schon vorhandene Komponenten aus dem Automobilbau und aus dem Bootsbau zurückgreifen können. Das Amphibienfahrzeug kann nicht aus einem vorhandenen Boot mit einem Auto zusammengesetzt werden, da diese in ihrer Funktionalität schon viel zu spezialisiert sind. Die beiden Einzelteile würden nicht zueinander passen, da sie in sich schon vollständig sind.

Ganz ähnlich verhält es sich mit implementierten Klassen. Die Implementierungen von Klassen sind genauso spezialisiert wie Produkte aus dem Maschinenbau. Sie passen nur dann zusammen, wenn sie als Komponenten für ein Teil-Ganzes-Verhältnis konstruiert wurden. Eine Vererbung sollte aber immer eine „Ist"-Beziehung modellieren; niemals eine Aggregation oder eine Komposition. Deshalb sollten in einer Mehrfachvererbung maximal eine Implementierung und ansonsten abstrakte Klassen vererbt werden.

Betrachtet man das Interfacekonzept in Java, so stellt man fest, dass die Java-Interfaces nichts anderes sind als die rein abstrakten Klassen in C++ ohne Implementierung. In Java wird der Schnittstellenklasse aus C++ einfach das sprachlich unterstützte Konzept des Interfaces entgegengesetzt. Das hat den wohlkalkulierten Grund, dass mit dem Zwang zur Verwendung der Mehrfachimplementierung von Interfaces – Mehrfachvererbung von Klassen ist ja nicht möglich – die Häufigkeit von Entwurfsfehlern sinkt. Der Java-Entwickler wird also dazu gezwungen, unproblematische Strukturen zu verwenden, selbst dann, wenn in Ausnahmefällen eine Abweichung von der Regel sogar gewinnbringend wäre. Diese Kontrolle durch die sprachliche Syntax gibt es in C++ nicht. C++ ist dadurch gewiss etwas mächtiger als Java. Die Konsequenzen dieser größeren Ausdrucksmöglichkeiten sind in den meisten Fällen allerdings problematisch, denn der C++-Entwickler muss mehr theoretisches Grundlagenwissen besitzen als der Java-Entwickler, um zu einem

guten Entwurf zu gelangen. Er wird schließlich nicht von der Sprache geleitet. Vor diesem Hintergrund kann es für einen C++-Entwickler durchaus sinnvoll sein, sich in der Java-Entwicklung einige Konzepte für seinen Bereich abzuschauen. Erfahrungsgemäß schärft sich damit das Bewusstsein für die Regeln des Softwareentwurfs.

Aber nun zur Syntax und zum Objektlayout bei der Anwendung der Mehrfachvererbung in C++:

Listing 2.131. Mehrfachvererbung

```
class A { /*...*/ };
class B { /*...*/ };

class X : public A, public B
{ /*...*/ };
```

In dem Beispiel wird die syntaktische Seite der Mehrfachvererbung gezeigt. Man trennt einfach mit einem Komma die Elternklassen voneinander ab, wobei jede ihren eigenen Sichtbarkeitsmodifikator besitzt. Es können beliebig viele Elternklassen für eine Kindklasse vererbt werden. Syntaktisch ist die Mehrfachvererbung also nur eine Erweiterung der einfachen Vererbung. Interessant wird die Mehrfachvererbung bei der Betrachtung des Objektlayouts. Nehmen wir einmal die folgende Situation an:

Listing 2.132. Mehrfachvererbung mit Attributen

```
class A
{
public:
  int a;
};

class B
{
public:
  int b;
};

class X : public A, public B
{ /*...*/ };
```

Das Layout der Klasse `X` setzt sich nun aus dem der Klassen `A` und `B` zusammen.

Wenn nun ein Zeiger des Typs `A*` auf ein Objekt des Typs `X` zeigen soll, so kann wie bisher ein einfacher Cast auf die Elternklasse durchgeführt werden. Wenn ein solcher Cast von `X*` auf den Typ `B*` stattfindet, so ändert sich auch der Zeigerwert. Schließlich muss ein Zeiger `B*` auch auf den `B`-Anteil des

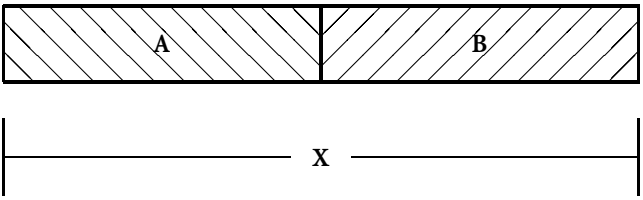


Abbildung 2.5. Objektlayout bei Mehrfachvererbung

`X`-Objektes verweisen. Es kann also nicht einfach eine syntaktische Neuinterpretation des Typs bei gleichem Zeigerwert stattfinden.

Listing 2.133. Adressverschiebung bei Upcast

```
// ...
int main()
{
    X x;
    A* pa = &x;
    B* pb = &x;

    std::cout << &x << std::endl;
    std::cout << pa << std::endl;
    std::cout << pb << std::endl;

    return 0;
}
```

Dieser Code veranschaulicht das Verhalten. Es werden die Adresswerte der entsprechenden Zeiger auf der Standardausgabe ausgegeben. Das Ergebnis kann beispielsweise folgendermaßen aussehen:

```
0012FF84
0012FF84
0012FF88
```

Die Adresswerte der Zeiger für das Objekt gleichen sich, wenn mit dem Typ `X*` oder `A*` auf das Objekt verwiesen wird. Nimmt man hingegen den Typ `B*`, so ändert sich der Adresswert im Beispiel um die Größe von `A`, dem Anteil von `X`, der vor dem `B`-Anteil liegt.

Für den Upcast existiert der implizite statische Cast. Für den Downcast muss entweder ein `static_cast<>()` oder ein `dynamic_cast<>()` durchgeführt werden, je nachdem, ob man eine Typenkontrolle zur Laufzeit wünscht oder nicht. Die Veränderung des Zeigerwertes ist auch im `static_cast<>()` enthalten, denn sie kann aus dem Klassenlayout berechnet werden.

Wichtig ist das Wissen um die Verschiebung der Zeiger dann, wenn man beabsichtigt, Adresswerte als eindeutige Objektidentifikationen zu verwenden[67]. Je nach betrachteter Schnittstelle hat ein Objekt eine andere Adresse.

2.2.33 Virtuelle Vererbung

Die virtuelle Vererbung ist eine Antwort auf ein Problem, das mit der Mehrfachvererbung auftritt. Im folgenden Beispiel wird eine Klasse `A` auf zwei Kindklassen `B1` und `B2` vererbt. In einem weiteren Schritt werden die beiden Kindklassen von `A` weitervererbt in die Klasse `C`. Die Mehrfachvererbung führt hier zur „Anhäufung" der Schnittstelle `A` in `C`.

Listing 2.134. Rautenförmige Vererbungshierarchie

```
class A { /*...*/ };

class B1 : public A { /*...*/ };
class B2 : public A { /*...*/ };

class C : public B1, public B2 { /*...*/ };

int main()
{
    B1 b1;
    C c;

    A *p1 = &b1;
    A *p2 = &c;   // Nicht erlaubt!

    return 0;
}
```

Die Konsequenz daraus ist, dass man mit einem Zeiger oder einer Referenz vom Typ `A` nicht mehr auf das Objekt `C` verweisen kann. Schließlich gibt es zwei `A`-Anteile in C und es müsste entschieden werden, auf welchen man mit dem Zeiger verweisen möchte. Das ist mit Hilfe eines zweistufigen Casts über eine `Bx`-Klasse auch möglich, wenngleich wenig ratsam.

Ein direkter Upcast zu `A` ist von `C` aus nur dann möglich, wenn `C` nur einen Anteil `A` besitzt. Das heißt aber auch, dass sich `B1` und `B2` ein Objekt `A` teilen müssten und keinen exklusiven Zugriff darauf hätten. Denkbar ist das vor allem dann, wenn `A` keine Implementierung enthält. Enthält sie Implementierung, so ist eine funktionale Verflechtung von den Kindklassen mit ihr eine

[67] Ein Grund mehr, warum man Adresswerte nie zur Bildung von IDs heranziehen sollte.

schon fast zwingende Konsequenz. Die Kindklassen `B1` und `B2` benötigten dann exklusive `A`-Anteile. Stellt `A` eine rein abstrakte Klasse dar, fungiert sie nur als Schnittstelle und kann gemeinsam „genutzt“ werden.

Diese gemeinsame Nutzung lässt sich mit dem Schlüsselwort `virtual` in der Klassenvererbung definieren.

Listing 2.135. Virtuelle Vererbung

```
class A { /*...*/ };

class B1 : virtual public A { /*...*/ };
class B2 : public virtual A { /*...*/ };

class C : public B1, public B2 { /*...*/ };

int main()
{
    B1 b1;
    C c;

    A *p1 = &b1;
    A *p2 = &c;

    return 0;
}
```

Die Stellung des Schlüsselwortes `virtual` ist mit dem Sichtbarkeitsbezeichner frei austauschbar, wie in dem Beispiel demonstriert wird. Wichtig ist aber, dass beide Klassen `B1` und `B2` die virtuelle Vererbung durchführen, denn sonst würde eine der Klassen einen exklusiven `A`-Anteil beanspruchen, womit wieder mehr als nur ein `A`-Objekt im `C`-Objekt enthalten wäre.

Interessant sind nun die Folgen für das Objektlayout einer Klasse, das virtuelle Anteile besitzt. Die Adresse des Anteils der virtuell geerbten Elternklasse steht nun nicht mehr in einer festen relativen Position gegenüber der Anfangsadresse des Objekts. Der virtuelle Elternklassenanteil muss über einen Zeiger in dem Objekt zur Laufzeit gefunden werden. Ein Upcast findet also über einen Zeiger statt, der implizit in das Objekt aufgenommen wurde. Der Upcast findet also nicht einfach über die Verschiebung des Adresswertes um einen festen Betrag statt. Die Verschiebung des Adresswertes geschieht indirekt über einen im Objekt befindlichen Zeiger. Das kann durchaus dazu führen, dass eine Konvertierung eines Typs a zu einem Typ b unterschiedliche Zeigerverschiebungen zur Folge hat.

Listing 2.136. Dynamische Verschiebung der Objektadresse beim dynamischen Upcast

```
#include <iostream>

class A { public: int a; };
class B1 : virtual public A { public: int b1; };
class B2 : public virtual A { public: int b2; };
class B3 : public virtual A { public: int b3; };

class C : public B1, public B2 { public: int c; };
class D : public C, public B3 {};

void f( C* pc );

int main()
{
    C c;
    D d;

    f( &c );
    f( &d );
    std::cout << ((A*)&d) << std::endl;

    return 0;
}

void f( C* pc )
{
    A *pa = pc;
    std::cout << pc << std::endl;
    std::cout << pa << std::endl;
}
```

Das Beispiel in Listing 2.136 demonstriert diese dynamische Adressverschiebung beim Upcast in einer virtuellen Vererbungsbeziehung. Die Ausgabe des Programms ist bei unterschiedlichen Compilern verschieden.

Eine Anforderung, die die virtuelle Vererbung an den Entwickler stellt, ist der Zwang Initialisierungslisten in folgenden Generationen zu wiederholen.

Listing 2.137. Initialisierung einer virtuellen Basisklasse

```
class A
{
public:
    A( int ) {}
};
```

```
class B : virtual public A
{
public:
    B() : A( 42 ) {}
};

class C : public B
{
public:
    // Der Konstruktor von A muss in der
    // Initialisierungsliste neu aufgerufen werden
    // obwohl er schon in der Initialisierung der
    // Klasse B aufgeführt wurde. Durch die virtuelle
    // Vererbung von A muss die Initialisierung
    // von A in jeder Kindklassengeneration neu
    // definiert werden!
    C() : A( 42 ) {}
};

int main()
{
    C c;

    return 0;
}
```

Da kein Objekt ein virtuell vererbtes Elternklassenobjekt exklusiv besitzt, kann es ein solches auch nicht exklusiv initialisieren. Im vorangehenden Beispiel wird die Klasse `B` von `A` abgeleitet. Der Konstruktor von `B` ruft in der Initialisierungsliste den Konstruktor von `A` auf. Die Kindklasse `C` von `B` kann diese Initialisierung von `A` nicht nutzen. `A` muss in der Initialisierungsliste von `C` neu initialisiert werden.

Eine virtuell vererbte Klasse erfordert, wenn sie keinen Standardkonstruktor hat, in jeder Kindklassengeneration eine explizite Initialisierung in der Initialisierungsliste der Konstruktoren.

2.2.34 Das Schlüsselwort `const`

Die Modellierung von Objektinteraktionen und die starke Typenkontrolle in C++ macht es notwendig, sich der Bedeutung des Schlüsselwortes `const` bewusst zu werden. Mit `const` zeigt man an, dass man etwas nicht verändern möchte bzw. kann. Lesezugriffe auf Daten werden aus in diesem Abschnitt

näher zu erläuternden Gründen maskiert. Bestimmte Operationen werden durch die Maskierung erst für tatsächlich konstante Daten ermöglicht. Es gibt mehr Bereiche für den Anwendungsbereich des Schlüsselwortes `const`, die nun systematisch beleuchtet werden sollen.

Wie im Abschnitt über die Konstanten 2.2.6 schon gezeigt, können mit `const` typisierte Konstanten deklariert werden.

```
const int maximum = 10000;
const double pi = 3.14159265358;
```

Dabei ist die Stellung des Schlüsselwortes vor oder nach dem Typenbezeichner gleichbedeutend.

```
int const maximum = 10000;
double const pi = 3.14159265358;
```

Gegenüber einer symbolischen Konstante, die mit `#define` durch den Präprozessor definiert wurde, haben die typisierten Konstanten den Vorteil, dass bei Bedarf durch den Compiler eine Konvertierung generiert werden kann, oder dass Fehler in der Typisierung durch die Syntaxüberprüfung auffallen und gut verständliche Meldungen nach sich ziehen.

Über die gerade angesprochene Stellung von `const` innerhalb einer Deklaration gibt es unterschiedliche Auffassungen darüber, welche angewendet werden sollte. Das Thema birgt ein paar mögliche Unsicherheiten, die bei der Aufzählung verschiedener Fälle von Deklarationen klar werden.

`const int antwort = 42;` deklariert eine einfache Konstante.
`int const antwort = 42;` ist die gleiche Konstante.

Bei der Deklaration von Zeigern gibt es die Varianten `const int *p;` oder `int const *p;`, was beides die Deklaration eines Zeigers auf konstanten Inhalt darstellt. Beachten Sie, dass der Zeiger als solcher variabel bleibt. Nur der Inhalt, auf den verwiesen wird, ist konstant oder wird durch den Zeiger maskiert – wie in Abschnitt 2.2.34 näher beschrieben.

Die Variante `int *const p = &x;` beschreibt die Deklaration eines konstanten Zeigers auf einen variablen Inhalt.

Lagern wir nun die eigentliche Beschreibung des Zeigers in eine Typendefinition aus, also `typedef int * INTPTR;`, kann ein Zeiger so definiert werden: `INTPTR p;`.

Wird dieser definierte Typ zusammen mit dem Schlüsselwort `const` verwendet, ergibt sich etwas, das zwar eindeutig definiert ist, manchmal aber Anlass zu einer Fehlinterpretation gibt: `const INTPTR p;`.

Da `INTPTR` ein Zeigertyp ist, ist `p` in diesem Fall ein konstanter Zeiger. Analog könnte `p` also als `int *const` deklariert sein. Der eine oder andere Entwickler könnte versucht sein, `p` als `const int *` zu verstehen, da scheinbar die gleiche deklarative Reihenfolge der Qualifizierer und des Typenbezeichners

zustande kam. Dabei muss man sich allerdings ins Gedächtnis rufen, dass `typedef` keine Textersetzungen durchführt, wie es durch `#define` geschieht, sondern den Typ als solchen mit einem neuen Namen belegt. Die falsche Interpretation hätte dann ihre Richtigkeit, wenn `INTPTR` so zustande gekommen wäre: `#define INTPTR int*`.

Um diese Quelle der unterschiedlichen Bedeutungen zu schließen, wird in dem Buch [C++-Templates] von N. Josuttis vorgeschlagen, den Qualifizierer `const` immer nachzustellen.

Die Deklaration `INTPTR const p;` wäre damit eindeutiger zu interpretieren. Allerdings sind die meisten C++-Quelltexte in der Weise geschrieben, die `const` voranstellt.

Die Const - Maskierung

Unter der Const-Maskierung ist zu verstehen, dass Zeiger- und Referenzparameter von Funktionen und Methoden mit dem Schlüsselwort `const` so modifiziert werden, dass auf die verwiesenen Daten kein schreibender Zugriff mehr möglich ist. Die Funktionen und Methoden erklären damit, dass sie die Daten, auf die sie Zugriff bekommen, nicht verändern werden. Der Compiler verbietet damit den Zugriff in den Funktionsrümpfen. Das hat noch einen weiteren Effekt. Solchen Parameterlisten können auch wirklich konstante Daten übergeben werden. Die Einschränkung der Zugriffsmöglichkeiten auf die in der Parameterliste übergebenen Daten führt zu einer breiteren Einsatzmöglichkeit der Funktion oder Methode selbst.

```
double Entfernung( const Punkt &p1, const Punkt &p2 );
double Entfernung( const Punkt *p1, const Punkt *p2 );
```

Oder auch so, wenn die Variante mit der nachgestellten Modifizierung angewendet werden soll:

```
double Entfernung( Punkt const &p1, Punkt const &p2 );
double Entfernung( Punkt const *p1, Punkt const *p2 );
```

Der Modifikator const an Klassenmethoden

Die Methoden, die keine attributären Änderungen an ihren Klassenobjekten durchführen, können auch als solche explizit deklariert werden. Dazu schreibt man hinter die Parameterliste, nach der schließenden Klammer, das Schlüsselwort `const`. Damit darf die Methode das eigene Objekt, für das sie aufgerufen wurde, nicht mehr ändern. Ebenso wie die Const-Maskierung in Parameterlisten ist die Anwendung des Modifikators `const` geboten, da nur solchermaßen deklarierte Methoden an konstanten Objekten aufgerufen werden können. Die mit `const` erklärte Einschränkung wirkt sich auch hier auf die Anwendbarkeit der Methode erweiternd aus.

Listing 2.138. Const-Maskierung

```
class Kreis
{
  public:
    Kreis( Punkt const &p, double radius );
    // ...
    double Umfang() const;
  private:
    Punkt m;
    double r;
};
// ...
double Kreis::Umfang() const
{
  return 2.0 * r * 3.141;
}
```

Sowohl Const-Maskierungen in Parameterlisten, wie auch die Modifikationen an Methoden durch `const` werden zur Unterscheidung überladener Methoden und Funktionen herangezogen. Jede Spezifikation mit `const` an der Methode oder in ihrer Parametersite verändert die Signatur der Methode.

Fehler in der Const-Maskierung und der Const-Modifikation können weitreichende Auswirkungen haben. Wenn in Basisklassen oder in einem Framework Fehler dieser Art enthalten sind – also keine korrekte Zugriffseinschränkung bei nur lesendem Zugriff –, können diese zu unangenehmen Typenkonvertierungen zwingen. Die Konvertierung `const_cast<>()` hat den Zweck, einen Ausweg aus einem vorliegenden Designfehler auszugleichen.

Wenn beipielsweise in der Klasse Kreis die Methode `Umfang()` nicht Const-modifiziert wäre und jemand die Klasse anwenden müsste, hätte er möglicherweise auch maskierte Zugriffe auf Kreis. Den Umfang könnte er dabei nicht berechnen, denn die Methode darf er nicht aufrufen, weil das zu einem Compilefehler führen würde.

Listing 2.139. Auswirkung der Const-Maskierung

```
void IrgendwelcheBerechnungen( Kreis const &k )
{
// ...
  double u = k.Umfang() // Nur erlaubt, wenn Umfang()
                        // const-modifiziert ist!
// ...
}
```

Man muss dabei berücksichtigen, dass mehrere Entwickler zusammenarbeiten, wobei nicht immer alles von jedem abgeändert werden kann. Klassen und ihre Methoden müssen also korrekt entworfen werden. Das heißt auch, dass lesende Zugriffe – Methoden – deutlich von den schreibenden unterschieden werden.

Konstante Rückgabewerte

Wenn Methoden Objekte zurückgeben, muss auf diese oft nur lesend zugegriffen werden. Wenn dies der Fall ist, können auch die Rückgabetypen von Methoden konstant deklariert werden. Ein Vorteil ergibt sich daraus, dass manche Compiler in bestimmten Fällen einen Kopiervorgang bei der Rückgabe des Objekts verhindern können. Als Beispiel kann ein Operator, der eine Addition zwischen zwei komplexen Zahlen ausführt, folgendermaßen definiert werden:

Listing 2.140. Konstanter Rückgabewert

```
const Complex operator+( const Complex &c1,
                         const Complex &c2 )
{
  return Complex( c1.r + c2.r, c1.i + c2.i );
}
```

2.2.35 Operatorüberladung

In C++ gibt es die Möglichkeit, Operatoren zu überladen. Es können beispielsweise arithmetische Operatoren für einen neuen, zusammengesetzten Zahlentyp definiert werden. So sind in der Mathematik bestimmte Operatoren, wie z. B. `+`, `--`, `*` und `/` für die komplexen Zahlen definiert. Will man das in C++ nachbilden, so muss man für einen Struktur- oder Klassentyp, der komplexe Zahlen repräsentieren kann, die entsprechenden Operatoren überladen.

Listing 2.141. Komplexe Zahl als Strukturtyp

```
struct Complex
{
  double r;
  double i;
};
```

Die Addition für komplexe Zahlen ist so definiert, dass einfach Real- und Imaginärteil separat addiert werden. Das soll an dieser Stelle einmal in C++ implementiert werden, um grundsätzlich zu zeigen, wie ein Operator überladen wird.

Listing 2.142. Überladener Operator

```
Complex operator+( const Complex &c1,
                   const Complex &c2 )
```

```
{
  Complex result;
  result.r = c1.r + c2.r;
  result.i = c1.i + c2.i;
  return result;
}
```

Ein Operator wird in C++ implementiert wie eine Funktion. Er ist eine Funktion, wobei sich der Funktionsname aus dem Schlüsselwort `operator` und dem entsprechenden Zeichen zusammensetzt. In der Parameterliste übernimmt er seine Operanden und das Ergebnis der Operation wird mit `return` zurückgegeben. Nun kann der Operator angewendet werden:

Listing 2.143. Anwendung eines überladenen Operators

```
...
Complex z1 = {3,5};
Complex z2 = {2,1};

Complex z3 = z1 + z2;
...
```

Operatoren werden nicht nur zu mathematischen Zwecken definiert. Wie in einigen Beispielcodes in diesem Buch schon vorgeführt wurde, arbeitet das Streamobjekt für die Standardausgabe `std::cout` mit dem Operator `<<`. In der Sprache selbst ist dieser Operator als Shiftoperator für Integertypen definiert. Für den Ausgabestrom hat er nicht nur neue Operanden bekommen, sondern auch eine ganz neue Interpretation. Wie ein solcher Operator für den Ausgabestrom definiert wird, soll anhand der Struktur `Complex` verdeutlicht werden.

Listing 2.144. Ausgabestreamoperator

```
std::ostream& operator<<( std::ostream &os,
                          const Complex &c )
{
  os << "(" << c.r << "/i" << c.i << ")";
  return os;
}
```

Die beiden Operanden des Operators unterscheiden sich vom Typ. Der rechte Operand ist das Objekt für den Ausgabestrom. Der linke ist das Objekt, das darauf ausgegeben werden soll, die Struktur `Complex`. Der Rückgabewert dient hier nicht zur Rückgabe eines Ergebnisses. Er ist dazu da, die Verkettungsmöglichkeit dieser Operatoren zu gewährleisten. Die Referenz, die er zurückliefert, dient einem eventuell nachfolgenden Operator als linker Operand. Hier der gesamte Code, der das Überladen von Operatoren und deren Verwendung veranschaulicht:

Listing 2.145. Anwendung überladener Operatoren

```
#include <iostream>

struct Complex
{
    double r;
    double i;
};

Complex operator+( const Complex &c1,
                   const Complex &c2 )
{
    Complex result;
    result.i = c1.i + c2.i;
    result.r = c1.r + c2.r;
    return result;
}
std::ostream& operator<<( std::ostream &os,
                          const Complex &c )
{
    return os << "(" << c.r << "/i" << c.i << ")";
}
int main()
{
    Complex z1 = {3,5};
    Complex z2 = {2,1};

    Complex z3 = z1 + z2;

    std::cout << z3 << std::endl;

    return 0;
}
```

Das Programm hat die Ausgabe:

```
(5/i6)
```

Wie schon angedeutet, ist der überladene Shiftoperator für die Ausgabe verkettbar. Zur Verkettbarkeit wird der Rückgabewert gebraucht. Der Operator hat den Rückgabetyp `std::ostream &`, der von einem eventuell nachfolgenden Operator als linker Operand gebraucht wird. Der für die Struktur `Complex` überladene Operator wird mit Hilfe schon bestehender Shiftoperatoren implementiert, die verkettet angewendet werden. Der neue Operator kann sich nun in solche Verkettungen einfügen.

Listing 2.146. Verkettung der überladenen Streamoperatoren

```
...
Complex z1 = {3,5};
Complex z2 = {2,1};

Complex z3 = z1 + z2;

std::cout << z1 << "+" << z2 << "=" << z3 << std::endl;
...
```

Der Compiler muss den richtigen Operator anhand der Typen der Operanden finden. Dazu formt er die Operatorschreibweise intern in eine Funktionsschreibweise um. Aus der Zeile

```
std::cout << z1;
```

wird die Form

```
operator<<( std::cout, z1 );
```

die auch eine gültige C++-Schreibweise ist[68]. Die Verkettung der Operatoren wird dann in eine Verschachtelung der Funktionen abgebildet, was die Verwendung des Rückgabewertes als linken Operanden des Folgeoperators verständlich macht.

Aus

```
std::cout << z1 << z2;
```

wird

```
operator<<( operator<<( std::cout, z1 ), z2 );
```

In dieser Umformung liegt das ganze Geheimnis der Operatorüberladung in C++. Operatoren können also als Funktionen gesehen werden wie andere auch. Sie unterliegen auch deren Regeln.

Funktionen können nun auch Elemente von Klassen sein. Man nennt sie dann Methoden. Also können auch Operatoren Elemente von Klassen sein. Sie unterliegen dann den gleichen Regeln, wie andere Klassenmethoden auch. Da aber Methoden mit Klassenattributen umgehen und dafür weniger Daten als Parameter übergeben bekommen, hat das auch Konsequenzen für das Aussehen von Operatoren, wenn sie als Klassenelemente definiert werden. Der linke Operand verschwindet aus der Parameterliste und das Objekt der Klasse selbst wird zum linken Operand. Dieser linke Operand muss auch ein konstantes Objekt sein können, weshalb der Operator die `const`-Spezifikation erhält.

[68] Unter bestimmten Umständen wird diese Funktionsschreibweise der Operatorschreibweise bevorzugt. Insbesondere dann, wenn in der Templateprogrammierung die Notwendigkeit zur Qualifizierung des Operators gegeben ist. Siehe Abschnitt 4.1.9 auf Seite 236.

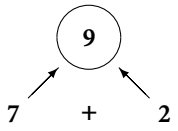
Listing 2.147. Überladener Operator als Klassenelement

```
class Complex
{
public:
  double r;
  double i;

  Complex operator+( const Complex &c ) const;
};

Complex Complex::operator+( const Complex &c ) const
{
  Complex result;
  result.r = r + c.r;
  result.i = i + c.i;
  return result;
}
```

Der Ausdruck `z1 + z2;` wird mit der alten, globalen Definition des Operators folgendermaßen umgeformt: `operator+(z1,z2);`. Mit der neuen Form, der Deklaration als Klassenelement, muss der Compiler einen Methodenaufruf generieren: `z1.operator+(z2);`.

Überladen arithmetischer Operatoren

Die arithmetischen Operatoren zeichnen sich dadurch aus, dass sie zwei Operanden haben. Die in C++ existierenden Bitoperatoren mit zwei Operanden (`&`, `|`, `^`, `<<` und `>>`) sollen der Einfachheit halber auch als arithmetische Operatoren aufgefasst werden. Ein Ausdruck mit einem beliebigen arithmetischen Operator bringt einen neuen Wert hervor. Die beiden Operanden bleiben dabei unverändert. Es wird also der Ergebniswert gewissermaßen durch den Operator erzeugt.



Nicht alle Operatoren haben eine solche Funktion. Der Zuweisungsoperator zum Beispiel verändert seinen linken Operand und hat einen Rückgabewert, nur damit er verkettet werden kann. In C++ wird man diesem Aspekt dadurch gerecht, dass innerhalb des Operators ein neues Objekt angelegt wird, das mit `return` nach außen geliefert wird. Wenn die arithmetischen und die Bitoperatoren semantisch nicht gänzlich umdefiniert werden[69], dann

[69] Im Fall des überladenen Shiftoperators « für den Ausgabestrom handelt es sich um einen semantisch umdefinierten Operator.

müssen sie einen Rückgabewert haben, der in ihrem Rumpf erzeugt wird. Der Rückgabetyp darf also keine Referenz sein, denn das lokale Objekt geht beim Verlassen des Operators verloren und die Referenz würde dann auf ein ungültiges Objekt verweisen. Das lokale Resultat muss als Kopie zurückgeliefert werden.

Listing 2.148. Kopie des Rückgabewertes

```
Complex operator+( const Complex &c1,
                   const Complex &c2 )
{
    Complex result = { c1.r + c2.r, c1.i + c2.i };
    return result;
}
```

oder

Listing 2.149. Kopie des Rückgabewertes

```
Complex Complex::operator+( const Complex &c ) const
{
    Complex result = { r + c.r, i + c.i };
    return result;
}
```

Für die anderen arithmetischen Operatoren gelten die gleichen Regeln. Also:

Listing 2.150. Überladung des Subtraktionsoperators

```
Complex Complex::operator-( const Complex &c ) const
{
    Complex result = { r - c.r, i - c.i };
    return result;
}
```

und

Listing 2.151. Überladung des Multiplikationsoperators

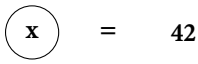
```
Complex Complex::operator*( const Complex &c ) const
{
    Complex result = { r*c.r - i*c.i, i*c.r + r*c.i };
    return result;
}
```

usw.

Überladen des Zuweisungsoperators

Jede Klasse und jede Struktur wird, wenn kein anderer definiert wird, mit einem Standard-Zuweisungsoperator ausgestattet, der eine einfache Bytekopie eines Objekts durchführt. Für viele Klassen, zumal für diejenigen, die eine Indirektion enthalten, kann es notwendig werden einen Zuweisungsoperator zu definieren[70].

Der Zuweisungsoperator hat eine ganz andere Umgebung als ein arithmetischer Operator. Der linke Operand eines Zuweisungsoperators wird verändert und es entsteht kein neuer Wert durch die Anwendung des Operators.



Zuweisungsoperatoren dürfen nur als Klassenelemente überladen werden. Da er den linken Operand – das Objekt also, an dem er aufgerufen wird – verändert, wird er nicht mit einer `const`-Spezifikation versehen.

Listing 2.152. Überladung des Zuweisungsoperators

```
class Complex
{
public:
  double i;
  double r;

  Complex& operator=( const Complex &c );
};

Complex& Complex::operator=( const Complex &c )
{
  i = c.i;
  r = c.r;
  return *this;
}
```

Der Rückgabewert hat die Aufgabe, den Operator verkettbar zu machen. In C++ ist die Schreibweise `a = b = c;` erlaubt. Damit der überladene Zuweisungsoperator in dieser Weise verwendet werden kann, gibt er eine Referenz zurück, die von einem weiteren Zuweisungsoperator als Operand verwendet werden kann.

[70] Es ist sogar empfehlenswert immer einen solchen zu deklarieren und wenn nötig auch zu implementieren, da der Standard-Zuweisungsoperator in vielen Fällen zu Fehlern führen kann, wenn man ihn einfach vergisst.

Dereferenzierungs- und Zugriffsoperatoren

Auch die Operatoren `*`, `[]` und `->` können überladen werden. Sinn ergibt das insbesondere im Zusammenhang mit Containern und Smart Pointern[71]. Aber auch andere Anwendungsbereiche für diese Operatoren sind denkbar. Gemeinsam ist den Operatoren `*` und `->`, dass sie nur einen Operanden haben und sich durch den Rückgabetyp auszeichnen. Das Symbol des Dereferenzierungsoperators `*` wird auch durch die Multiplikation in Anspruch genommen. Um also den Stern als Dereferenzierungsoperator zu überladen, muss man ihn mit nur einem Operanden definieren. Im nachfolgenden Listing wird der Operator als Klassenelement definiert, was ihm einen Parameter – Operanden – implizit einträgt. Der Operand ist also nicht in seiner Parameterliste erkennbar, sondern wird in Form des `this`-Zeigers an den Operator übergeben.

Listing 2.153. Der überladene Dereferenzierungsoperator

```
class FunctionalProxy
{
public:
  // ...
  Something& operator*();
private:
  Something *element;
};

Something& FunctionalProxy::operator*()
{
  // ...
  return *element;
}
```

Würde man ihn global definieren, würde er folgendermaßen aussehen:

Listing 2.154. Globaler Dereferenzierungsoperator

```
class FunctionalProxy
{
public:
  // ...
  friend Something& operator*(FunctionalProxy &);
private:
  Something *element;
};

Something& operator*(FunctionalProxy &p)
```

[71] Siehe Abschnitt 5.2.9 auf Seite 342.

```
{
  // ...
  return *(p->element);
}
```

Der Pfeiloperator kann ganz ähnlich implementiert werden wie der Sternoperator. Allerdings kann man ihn nicht global definieren.

Listing 2.155. Überladener Pfeiloperator

```
class FunctionalProxy
{
public:
  // ...
  Something* operator->();
private:
  Something *element;
};

Something* FunctionalProxy::operator->()
{
  // ...
  return element;
}
```

In den folgenden Abschnitten unter 3.3 werden die Stern- und Pfeiloperatoren für Zugriffe auf Datencontainer überladen, im Abschnitt 4.1.15 finden die Operatoren eine Anwendung auf einen Smart Pointer.

Überladen von Typenkonvertierungsoperatoren

Neben den Konstruktoren mit einem Parameter gibt es eine weitere Gruppe von Methoden, die Typenkonvertierungen definieren: die Typenkonvertierungsoperatoren. Sie werden aber nicht im Zieltyp definiert, sondern im Quelltyp.

Ihre Syntax hat eine spezielle Form. Sie sind immer Elemente der Klasse, die in einen Zieltyp konvertiert werden soll. Man definiert keinen Rückgabetyp, denn der Operator an sich hat ja den Sinn, einen bestimmten Wert zu formen. Der Typ steckt im Namen des Operators. Die allgemeine Form ist die folgende:

Listing 2.156. Allgemeine Form eines Typenkonvertierungsoperators

```
class Quelltyp
{
  // ...
  operator Zieltyp();
};
```

Wenn z. B. ein Operator definiert werden soll, der nach `bool` konvertiert, dann sieht das so aus:

Listing 2.157. Typenkonvertierungsoperator nach `bool`

```
class X
{
  // ...
  operator bool();
};
```

Speziell die Operatoren, die nach `bool` konvertieren, werden häufig zur Überprüfung der Gültigkeit des Objektzustandes zur Verfügung gestellt. Die Gültigkeit kann damit in der Form `if( obj ) ...` abgefragt werden. Natürlich können auch Operatoren für alle anderen Standardtypen und für beliebige Klassentypen definiert werden. Wichtig ist dabei, dass Konvertierungen nicht doppelt definiert werden. Denn es ist ja möglich, dass im Zieltyp schon ein entsprechender Konstruktor existiert.

Das Überladen von Klammeroperatoren

Es können sowohl die runden Klammern `()`, wie auch die eckigen `[]` überladen werden. Da die eckigen Klammern zur Indizierung von Arrays verwendet werden, spricht man bei ihnen auch vom Indexoperator. Häufig wird dieser Operator im Zusammenhang mit einem indizierten Containerzugriff überladen.

Listing 2.158. Überladener Indexoperator

```
class Vector
{
  //...
  ELEMENT operator[]( unsigned idx ) const;
};
```

Der Indexierungsoperator `[]` hat einen Operanden (Parameter) und kann aber nur als Klassenelement implementiert werden. Eine globale Deklaration ist nicht möglich. Eine Implementierung wird in dem Abschnitt 3.3.2 auf Seite 208 für eine Vektorimplementierung gezeigt. In der Standardbibliothek gibt es viele sinnvolle Überladungen für verschiedene Containertypen, wie z. B. für `std::vector` und `std::valarray`.

Die runden Klammern haben einen ganz anderen Anwendungsbereich. Mit ihnen kann ein Objekt das syntaktische Verhalten eines Funktionszeigers gewinnen. In der STL und in anderen Bereichen der Standardbibliothek werden solche „Funktoren“ angewendet. Es ist auch nicht schwer selbst Funktoren zu schreiben. Im Grunde genommen stellt die Möglichkeit, den Klammeroperator zu überladen, eine ideale Möglichkeit zur Implementierung des

Strategiemusters und des Befehlsmusters aus [GoF95] dar. Anstatt der Methode `execute()` im Command Pattern kann einfach der Klammeroperator überladen werden.

Listing 2.159. Überladener Klammeroperator

```
class Command
{
public:
  // ...
  virtual bool operator()() = 0;
};
```

Da der Operator als Klassenelement definiert ist, kann er natürlich auch virtuell sein. Der Klammeroperator muss übrigens immer als Klassenelement definiert werden. Er kann beliebige Parameterlisten entgegennehmen.

2.2.36 Exception Handling

Das Exception Handling, zu deutsch Ausnahmebehandlung, ist ein Konzept, das erst mit der ANSI/ISO-Standardisierung in C++ aufgenommen wurde. Es wirkt einerseits strukturierend auf die Behandlung von Ausnahmen, die zur Laufzeit stattfinden, andererseits erschwert es den Zugang zur C++-Entwicklung durch zusätzliche Komplexität. Man kann nicht einfach einen standardkonformen C++-Compiler einsetzen, ohne sich zumindest am Rande mit dem Thema Ausnahmebehandlung zu befassen[72].

Beginnen wir mit einem Beispiel, das wir im Abschnitt 2.2.23 zur Demonstration des Operators `new` kennengelernt haben.

Listing 2.160. Exception bei fehlgeschlagener Allokation

```
#include <iostream>

class X
{
public:
  X() { std::cout << "X::X()" << std::endl; }
  ~X() { std::cout << "X::~X()" << std::endl; }
};

int main()
{
  try
```

[72] Und sei es nur, um das Exception Handling so gut es geht zu deaktivieren oder zu umgehen.

```
  {
    X *p = new X;

    delete p;
  }
  catch( std::bad_alloc &e )
  {
    // Fehlerfall
  }

  return 0;
}
```

Wenn die Allokation fehlschlägt, wirft der Operator `new` eine Ausnahme vom Typ `std::bad_alloc`. Die Schlüsselworte, die im Zusammenhang mit der Ausnahmebehandlung stehen, sind `try`, `catch` und in diesem Beispiel noch unsichtbar `throw`. Das Paar `try` und `catch` definiert einen so genannten *Exceptionhandler*. Im `try`-Block stehen die Anweisungen, die den Standardfall des Programmlaufs darstellen und die im Ausnahmefall eine Exception werfen. Wenn eine Exception geworfen wird, wird die Ausführung des `try`-Blocks sofort unterbrochen. Anweisungen, die nach der Stelle stehen, die die Ausnahme ausgelöst hat, werden nicht mehr ausgeführt. Stattdessen wird der Stack bis auf die Ebene abgebaut, die der Umgebung des `try`-Blocks entspricht. Beim Abbau des Stacks werden alle Destruktoren der lokalen Objekte korrekt aufgerufen. Danach wird für die Ausnahme ein passender Handler gesucht. Das heißt, dass die Ausführung des Progamms in dem `catch`-Block weiterläuft, der den Typ der Exception oder einen Basistyp ausweist. Es können mehrere `catch`-Blöcke existieren, die alle unterschiedliche Exceptiontypen aufweisen. Um dies zu zeigen und auch um die werfende Seite zu demonstrieren, sei das folgende einfache Beispiel gezeigt:

Listing 2.161. Mehrere `catch`-Blöcke

```
#include <iostream>
using namespace std;

void fa()
{
  cout << "Aufruf von fa()" << endl;
}

void fb()
{
  cout << "Aufruf von fb()" << endl;
  throw int(42);
}
```

```
void fc()
{
  cout << "Aufruf von fc()" << endl;
}

int main()
{
  try
  {
    fa();
    fb();
    fc();
  }
  catch( double d )
  {
    cout << "Ausnahme vom Typ double" << endl;
    cout << "Wert: " << d << endl;
  }
  catch( int i )
  {
    cout << "Ausnahme vom Typ int" << endl;
    cout << "Wert: " << i << endl;
  }
  return 0;
}
```

Die `catch`-Blöcke werden also anhand von Typen unterschieden. An der Ausgabe des Beispielprogramms ist auch zu sehen, dass die Funktion `fc()` innerhalb des `try`-Blocks nicht mehr aufgerufen wird:

```
Aufruf von fa()
Aufruf von fb()
Ausnahme vom Typ int
Wert: 42
```

Geworfen wird die Exception mit der `throw`-Anweisung. Mit dieser Anweisung stoppt die weitere Ausführung des Programms und das beschriebene Prozedere der Stackbereinigung und des Sprungs in den `catch`-Block. Dabei kann sich die `throw`-Anweisung beliebig tief in einer Aufrufhierarchie verstecken. Der Stack wird bis zu der Ebene abgebaut, wo der entsprechende Handler zu finden ist.

Listing 2.162. Hierarchische Exceptionhandler

```
#include <iostream>
using namespace std;
```

```
class EA {};
class EB {};

void e()
{
  throw EA();
}

void f()
{
  try
  {
    e();
  }
  catch( EB &e )
  {
    cout << "Die Exception EB ist aufgetreten!" << endl;
  }
}

int main()
{
  try
  {
    f();
  }
  catch( EA &e )
  {
    cout << "Die Exception EA ist aufgetreten!" << endl;
  }

  return 0;
}
```

In der Aufrufreihenfolge des Beispiels ruft Funktion `f()` die Funktion `e()`. Dabei stellt `f()` einen Handler (`try-catch`-Block) zur Verfügung, der aber auf die durch `e()` geworfene Exception nicht passt. Dadurch fällt die Exception an der Stackposition des Handlers vorbei, baut den Stack also weiter nach unten ab, bis sie schließlich den passenden Handler in der Funktion `main()` findet. Dort wird sie im entsprechenden `catch`-Block gefangen. Man kann sagen, dass die Exceptions gewissermaßen einen kontrollierten Absturz darstellen. Die Unterscheidung anhand von Typen ermöglicht eine Klassifizierung der Ausnahmen. Ausnahmen werden also üblicherweise in Klassenform definiert. Die Verwendung von Standarddatentypen wie im gezeigten Beispiel

ergibt normalerweise wenig Sinn. Neben dieser Idee der begrifflichen Unterscheidung gibt es noch ein anderes gutes Argument für die Verwendung von Klassen zur Definition von Ausnahmen. Eine Klasse kann Werte aufnehmen, um Informationen über den Kontext der aufgetretenen Ausnahme an die Stelle zu liefern, wo die Ausnahme bearbeitet wird.

Listing 2.163. Ausnahmebehandlung anhand der Typunterscheidung

```
class Ausnahme1{};
class Ausnahme2{};
class Ausnahme3{};
class AusnahmeX{};

void f1()
{
// ...
  throw Ausnahme1();
// ...
}
void f2()
{
// ...
  throw Ausnahme2();
// ...
}
void f3()
{
// ...
  throw Ausnahme3();
// ...
}

int main()
{
  try
  {
    f1();
    f2();
    f3();
  }
  catch( Ausnahme1 &e1 )
  {
    cout << "Die Ausnahme 1 ist aufgetreten!" << endl;
  }
  catch( Ausnahme2 &e2 )
```

```
  {
    cout << "Die Ausnahme 2 ist aufgetreten!" << endl;
  }
  catch( Ausnahme3 &e3 )
  {
    cout << "Die Ausnahme 3 ist aufgetreten!" << endl;
  }
  catch( ... )
  {
    cout << "Eine unbekannte Ausnahme ist aufgetreten!"
         << endl;
  }

  return 0;
}
```

Das Beispiel zeigt schematisch, wie Ausnahmen durch ihre `catch`-Blöcke gefangen werden. Dabei gibt es auch den `catch`-Block, der alle nicht weiter spezifizierten Ausnahmen fängt. Da die Ausnahme in ihm nicht typisiert ist, kann auch nicht auf sie zugegriffen werden. Man verwendet den unspezifizierten `catch`-Block vor allem dafür, alle möglichen Exceptions in einem Handler garantiert abzufangen, damit keine von ihnen nach außen dringt. Man kann die `throw`-Anweisung auch dazu benutzen, eine Exception weiter zu werfen. In diesem Fall möchte man vielleicht eine Teilbehandlung im aktuellen `try-catch()`-Block vornehmen und die weitere Behandlung einem übergeordneten Exceptionhandler überlassen.

Listing 2.164. Weiterreichen einer teilbehandelten Ausnahme

```
// ...
catch( Ausnahme1 &e1 )
{
  cout << "Die Ausnahme 1 ist aufgetreten!" << endl;
  throw e1;
}
// ...
```

In der Standardbibliothek gibt es eine Basisklasse für Exceptions, die eine virtuelle Methode zur Reimplementierung anbietet, die eine Textausgabe mit Informationen über die Ausnahme zurückgibt. Die Klasse heißt `std::exception`, die Methode `what()`. Im folgenden Beispiel finden Sie eine einfache Demonstration der Anwendung der Basisklasse `std::exception`.

Listing 2.165. Implementierung der Methode `what()`

```
#include <iostream>
#include <string>
```

```
class MyException : public std::exception
{
public:
  MyException( char *where ) : txt(where) {}
  const char* what() const throw();
private:
  char *txt;
};

const char* MyException::what() const throw()
{
  static std::string s;
  s = "MyException ist in ";
  s += txt;
  s += " aufgetreten.";
  return s.c_str();
}

void f()
{
  throw MyException( "f() in Zeile soundso" );
}

int main()
{
  try
  {
    f();
  }
  catch( MyException &e )
  {
    std::cerr << e.what() << std::endl;
  }

  return 0;
}
```

Natürlich darf eine Ausnahmeklasse weitere Daten und Methoden aufnehmen, um etwas über den Kontext und den Ursprung der Ausnahme auszusagen. Wie Ausnahmeklassen zu definieren sind, damit befasst sich der Abschnitt 2.2.36. Zunächst soll aber näher betrachtet werden, was sonst im Zusammenhang mit der Verwendung von Exception Handling in einem Projekt zu beachten ist.

Im vorangegangenen Beispiel der Nutzung der Basisklasse `std::exception` zur Definition einer eigenen Ausnahmeklasse hat die Methode `what()` eine Deklaration, in der das Schlüsselwort `throw` steht. Diese `throw`-Deklaration erklärt, dass die Methode `what()` keine Exception wirft, denn die Klammern hinter `throw` sind leer. In den Klammern können Typen stehen, die für eine Methode als Ausnahmen zugelassen sind. Mehrere Typen werden dabei durch Komma getrennt. Wirft eine Methode eine Ausnahme, die nicht in ihrer `throw`-Deklaration auftaucht, so ist das Verhalten undefiniert. In den meisten Fällen wird das Programm durch den Aufruf der Funktion `terminate()` aus der Standardbibliothek beendet. Methoden, die keine `throw`-Deklaration haben, dürfen alle Ausnahmen werfen, ohne dass das Programm beendet wird. Wer die Sprache Java kennt, wird an dieser Stelle vielleicht etwas enttäuscht sein über die Regeln in C++. In C++ kann korrektes Exception Handling nicht durch den Compiler überprüft und garantiert werden. Man muss in C++ zu fast jeder Zeit mit dem Auftreten einer Exception rechnen. Das zieht bestimmte Notwendigkeiten für den Softwareentwurf nach sich.

Regeln für das Schreiben von Exception - sicherem Code

Bedenkt man, dass in einem C++-Programm zu fast jeder Zeit eine Ausnahme geworfen werden kann, so muss der Code in einer besonderen Weise geschrieben werden, damit er nicht in undefinierte Zustände gerät. Bestimmte Exception-freie Abschnitte müssen von solchem Code separiert werden, in dem Exceptions auftreten können. Es muss vor allem eine bestimmte Struktur eingehalten werden, die man als Exception-sicher bezeichnen kann. Es gibt auch bestimmte Dinge, die man als generell unsicher bezeichnen muss und die daher zu vermeiden sind. Dazu gehören vor allen anderen Dingen die typischen Anweisungspaare von Allokation und Freigabe irgendwelcher Systemressourcen:

Listing 2.166. Allokation von Speicher

```
//...
char *p = (char*)malloc( X * sizeof(char) );

fkt();

free( p );
//...
```

Oder:

Listing 2.167. Öffnen von Dateien

```
//...
FILE *f = fopen( "daten.txt", "wt" );

fkt();
```

```
fclose( f );
//...
```

Wenn zwischen den Aufrufen solcher typischer Funktionspaare eine Exception auftritt, wird die schließende bzw. freigebende Funktion nicht mehr aufgerufen. Systemressourcen bleiben damit belegt oder werden nicht freigegeben. Auch die Bearbeitung der eigentlichen Ausnahme kann daran nichts ändern. Eine solche Struktur führt also dazu, dass eine Ausnahme neue Probleme zur Laufzeit verursacht. Damit solche freigebenden Codeteile auch nach dem Auftreten einer Exception aufgerufen werden, muss man sich das Stackunwinding zu Nutze machen. Die Exception fällt auf die Ebene des Stacks zurück, wo sich eine Behandlung für sie findet. Der Stack der dabei passiert wird, wird korrekt abgebaut. Das heißt auch, dass für alle Objekte, die oberhalb der Ausnahmebehandlung liegen, der Destruktor aufgerufen wird. Dieser Destruktoraufruf ist der Schlüssel zur Lösung des beschriebenen Problems. Die beschriebenen Funktionspaare zur Allokation und Deallokation von Systemressourcen müssen in die Konstruktoren und Destruktoren von Objekten. Etwas schematisch skizziert, könnte es so aussehen:

Listing 2.168. Eine einfache Dateiklasse

```
class Datei
{
public:
  Datei();
  ~Datei();
  // ...
private:
  FILE f;
  Datei( const Datei& );
};

Datei::Datei()
{
  f = fopen( "daten.txt", "wt" );
}

Datei::~Datei()
{
  fclose( f );
}
```

Dort wo eine solche Datei verwendet werden soll, legt man ein Objekt der Klasse auf dem Stack an:

```
   Datei d;
```

Wenn nun in der Folge eine Ausnahme auftritt, wird das Objekt korrekt mit einem Destruktoraufruf vom Stack entfernt, was dazu führt, dass auch `fclose()` aufgerufen wird. Mit allen Funktionspaaren dieser Art muss so umgegangen werden, dass sie in Klassen ausgelagert werden.

Dabei können Konstruktoren Ausnahmen werfen, wenn sie fehlschlagen. Das Objekt wird erst dann auf dem Stack vollständig angelegt, wenn der Konstruktor durchlaufen wurde. Wenn er vorher eine Ausnahme wirft, wird der Speicherplatz auch gleich wieder freigegeben.

Listing 2.169. Abbruch des Konstruktors mit einer Exception

```
Datei::Datei()
{
  f = fopen( "daten.txt", "wt" );
  if( f == NULL ) throw DateiFehler();
}

Datei::~Datei()
{
  fclose( f );
}
```

Natürlich stellt die Standardbibliothek Klassen zur Verfügung, die das Arbeiten mit Dateien ermöglichen. Man muss sich also nur bedienen, denn die schematisch gezeigte Lösung steht schon mit viel größerem Komfort zur Verfügung. Im Abschnitt 5.2.4 auf Seite 289 werden die Klassen zur Bearbeitung von Dateien beschrieben.

Der Destruktor darf übrigens niemals eine Ausnahme werfen. Schließlich könnte er gerade in Folge einer Exception im Rahmen des Stackunwindings aufgerufen werden. Würde er nun eine Exception werfen, gäbe es keine Möglichkeit mehr der Reaktion darauf. Es würde sofort die Methode `terminate()` aus der Standardbibliothek aufgerufen werden. Im Falle des Aufrufs von `fclose()` kann davon ausgegangen werden, dass keine Exception geworfen wird. Die Funktion kommt aus der Stadardbibliothek von C und ist dort ohne Exception definiert[73]. Man muss also in einem solchen speziellen Fall nichts Besonderes unternehmen. Wenn jedoch Funktionen im Destruktor aufgerufen werden, die potentiell Exceptions werfen können, muss man den Destruktor sichern. Das geschieht dadurch, dass man eine `try-catch`-Behandlung in den Destruktor aufnimmt. Dieser Handler braucht auch einen `catch`-Block mit nicht spezifizierter Exception `catch( ... )`, damit auch wirklich alle möglichen Exceptions gefangen werden und keine nach außen tritt. Schematisch kann man das so darstellen:

[73] Nicht zuletzt deshalb, weil C kein Exception Handling kennt.

Listing 2.170. Schematische Darstellung eines Exception-sicheren Destruktors

```
T::~T()
{
  try
  {
    Anweisung1;
    Anweisung2;
    // ...
  }
  catch( ... )
  {
  }
}
```

Die Regel lautet also, dass Destruktoren niemals Exceptions werfen dürfen. Auf so etwas kann es keine adäquate Reaktion mehr geben. Kommen wir nun zu den Konstruktoren:

Exceptions in Konstruktoren

Der Operator `new` hat mit der ANSI/ISO-Standardisierung von C++ ein ganz neues Verhalten für den Ausnahmefall bekommen. Statt wie bisher in den AT&T-Standards einen Nullzeiger zurückzuliefern, wirft `new` eine Exception. In einigen vorangegangenen Beispielen wurden die beiden Varianten des Operators schon gegenübergestellt. Der aus der Anwendung von `new` resultierende Code sieht in diesen beiden Varianten sehr unterschiedlich aus. Für den Programmierer hat sich also mit ANSI/ISO-C++ strukturell sehr viel geändert. Wir haben zunächst den Allokationsvorgang von außen betrachtet und gesehen, wie der Fehlschlag einer Speicheranforderung durch `new` in einem `try-catch()`-Konstrukt abgefangen wird. Die Konsequenzen für die Definition eines Konstruktors selbst haben wir noch nicht betrachtet. Genau dies soll in den folgenden Abschnitten geschehen. Wir befassen uns hier mit dem Aufbau eines Konstruktors, der den neuen Gegebenheiten von ANSI/ISO-C++ gerecht wird.

Üblicherweise allozieren die Konstruktoren Speicher für das Objekt, legen abhängige Objekte auf dem Heap an und führen andere Operationen aus, die alle einzeln für sich fehlschlagen können. Man kann ganz allgemein formuliert sagen, dass Konstruktoren Systemressourcen allozieren. Systemressourcen sind endlich, woraus folgt, dass deren Allokation fehlschlagen kann.

Als Beispiel sei eine Klasse angeführt, die die Aufgabe hat, eine Logdatei zu schreiben. Eine solche Klasse öffnet eine Datei – die Logdatei, in die entsprechende Einträge geschrieben werden sollen. Diese Klasse braucht wahrscheinlich auch einen Puffer im Speicher, damit während des Schreibens keine Allokationen stattfinden müssen. Würde ein Logeintrag, der besagt, dass kein

Speicher mehr im System vorhanden ist, zusätzliche Allokationen verursachen, könnte er möglicherweise nicht mehr in die Datei geschrieben werden. Deshalb ist die Vorallokation im Konstruktor der Logdateiklasse notwendig. Die Klasse benötigt also zwei unabhängige Systemressourcen: einen Speicherblock und eine Datei. Beide Systemressourcen können durch das Betriebssystem aus unterschiedlichen Gründen verweigert werden. Dabei kann sowohl das Öffnen der Datei verweigert werden wie auch die Allokation des Speichers. Neben diesen zwei Fällen gibt es noch den dritten Fall, dass beide Ressourcen nicht zur Verfügung stehen. Der Standardfall steht also drei Ausnahmefällen gegenüber. In jedem der Ausnahmefälle muss anders reagiert werden.

Listing 2.171. Implementierung einer Logfile-Klasse mit Exceptions

```
#include <iostream>
#include <memory>
#include <string>
#include <stdexcept>
#include <cstdio>
#include <strstream>
using namespace std;

class Logfile_exception : public std::exception
{
public:
   Logfile_exception()
   : std::exception(), txt("logfile exception") {}
   Logfile_exception( const Logfile_exception& e )
   : std::exception( e ), txt("logfile exception") {}
   const char *what( ) const throw();
private:
   const char *const txt;
};

class Logfile
{
  public:
    Logfile(char *fname)
    throw(std::bad_alloc, Logfile_exception);
    ~Logfile() throw();

    void write( char *str );
    Logfile& operator<<( char *str )
    { write( str ); return *this; }
    Logfile& operator<<( int i );
```

```
  private:
    Logfile( const Logfile& ); // to hide!
    FILE *f;
    char *txt;
};

const char * Logfile_exception::what( ) const throw()
{ return txt; }

Logfile::Logfile(char *fname)
throw(std::bad_alloc, Logfile_exception)
: txt( new char [1024] )
{
  f = fopen( fname, "wt" );
  if( ! f )
  {
    delete [] txt;
    throw Logfile_exception();
  }
}

Logfile::~Logfile() throw()
{
  delete [] txt;
  fclose( f );
}

void Logfile::write( char *str )
{
  fprintf( f, str );
}

Logfile& Logfile::operator<<( int i )
{
  txt = "\0";
  std::strstream str( txt, 1024 );
  str << i;
  str << "\0";
  write( txt );
  return *this;
}
```

```
int main()
{
  try
  {
    Logfile l( "logfile.txt" );
    l << "Einen Testeintrag\n";
    l << 42;
  }
  catch( std::bad_alloc &e )
  {
    cerr << e.what() << endl;
  }
  catch( Logfile_exception &e )
  {
    cerr << e.what() << endl;
  }

  return 0;
}
```

Die verschiedenen Ausnahmefälle fordern bei dieser herkömmlichen Allokation von Systemressourcen eine Behandlung innerhalb des Konstruktors. Je mehr Allokationen für eine Klasse durchgeführt werden müssen, desto aufwändiger wird die Behandlung der Ausnahmen innerhalb des Konstruktors, bevor eine Exception geworfen werden kann. Das ist zwar grundsätzlich machbar und auch nicht falsch im eigentlichen Sinne, es ist jedoch fehleranfällig und wenig elegant.

Listing 2.172. Schema einer Fehlerbehandlung in einem Konstruktor

```
Logfile::Logfile(char *fname)
throw(std::bad_alloc, Logfile_exception)
: txt( new char [1024] )
{
  f = fopen( fname, "wt" ); // Standardfall
  if( ! f )                 // 1. Zeile Fehlerbehandlung
  {                         // 2. Zeile Fehlerbehandlung
    delete [] txt;          // 3. Zeile Fehlerbehandlung
    throw Logfile_exception(); // 4. Zeile Fehlerb.
  }                         // 5. Zeile Fehlerbehandlung
}
```

Um strukturell Abhilfe zu schaffen, gibt es in der Standardbibliothek den Autopointer `std::auto_ptr<T>`. Die genaue Funktionalität des Autopointers ist im Abschnitt 5.2.9 ab Seite 342 beschrieben. An dieser Stelle soll der Autopointer nur in seiner hauptsächlichen Funktion erklärt werden:

Listing 2.173. Sicherung dynamisch allokierter Objekte durch Autopointer

```
#include <iostream>
#include <memory>

class X
{
public:
  X() { std::cout << "X::X()" << std::endl; }
  ~X() { std::cout << "X::~X()" << std::endl; }
private:
  X( const X& );
};

int main()
{
  std::auto_ptr<X> p1 ( new X() );

  return 0;
```

In dem Beispiel wird ein Objekt der Klasse `X` dynamisch mit Hilfe des `new`-Operators instanziiert. Mit dem Autopointer kann auf dieses Objekt verwiesen werden. Wenn der Autopointer vom Stack entfernt wird, löscht er auch das dynamische Objekt. Dieses Verhalten lässt sich für Klassen nutzen, die abhängige Objekte instanziieren und Exception-sicher sein müssen. Wenn man innerhalb der Klasse nicht die Zeiger auf die aggregierten Objekte, sondern Autopointer darauf deklariert, werden diese auch im Exceptionfall gelöscht.

Einige Regeln zur Definition von Ausnahmeklassen

1. Die eigene Ausnahmeklasse sollte von der Standardklasse `exception` im Namensraum `std` abgeleitet werden. Darauf sollte nur in speziellen Fällen verzichtet werden, in denen die Ausstattung der Klasse mit einer V-Tabelle zu teuer ist. Die Beachtung dieser Regel versetzt andere Entwickler in die Lage, ihre Ausnahme zusammen mit den anderen durch `catch(std::exception &e)` abzufangen. Die Verwendung des Default Catch Blocks `catch( ... )` hat den Nachteil, dass kein Ausnahmeobjekt abgefragt werden kann.
2. Es sollte bei der Ableitung von `exception` immer die virtuelle Vererbung eingesetzt werden. Vielleicht wird ihre Ausnahmeklasse zu einer Basisklasse einer spezielleren Ausnahmeklasse, die zusätzlich noch von weiteren Ausnahmebasisklassen erbt. Wenn bei der Ableitung von `exception` keine virtuelle Vererbung eingesetzt wurde, greift der Catch Block für die Standardausnahme `catch(std::exception &)` nicht mehr, da der Upcast der Referenz zweideutig ist.

3. Innerhalb einer Ausnahmeklasse dürfen sich keine Datenelemente befinden, die ihrerseits bei der Instanziierung eine Ausnahme werfen können (insbesondere der Kopierkonstruktor ist ein gefährlicher Kandidat). So können beispielsweise die Konstruktoren der Klasse `std::string` die Ausnahme `std::out_of_range` und `std::bad_alloc` werfen. Vor allem die letztere ist gefährlich, da sie in einem Fall auftritt, der womöglich gerade durch ihre Ausnahme behandelt werden soll. Um Daten in Ausnahmeklassen zu transportieren, sollten also Typen verwendet werden, die selbst keine Ausnahmen werfen[74].
4. Die Methode `what()` in ihrer Ausnahmeklasse soll einen Text liefern, der etwas über die aufgetretene Ausnahme und deren Kontext aussagt. Dieser Text sollte immer in der Methode zusammengebaut werden – auf keinen Fall schon im Konstruktor der Klasse. Textoperationen sind potentielle Auslöser von Ausnahmen irgendwelcher Art, da sie den Speicher dynamisch nutzen. Wenn eine Ausnahme auftritt, dann sollte dies nach dem Abbau des Stacks geschehen. Außerdem kann innerhalb der Methode `what()` ein `try-catch`-Block eingebaut werden, um eine nach außen dringende Exception zu verhindern. Schließlich ist sie definiert als:

   ```
   virtual const char* what() const throw();
   ```

   Wenn dies möglich ist, ist die Nutzung statischer Strings zu empfehlen, da dann keine Speicheroperationen während der Abarbeitung der Ausnahme mehr stattfinden müssen.
5. Denken Sie an die Definition eines Kopierkonstruktors! Wenn jemand im Catch-Block den Ampersand `&` für die Referenzdeklaration vergisst, wird das Ausnahmeobjekt beim Fangen kopiert. Es gibt sogar Compiler, die das Ausnahmeobjekt immer kopieren[75]. Die Technik mit dem privaten Kopierkonstruktor funktioniert bei solchen Compilern also nicht. Außerdem gibt es mindestens einen Compiler, der beim Werfen und Kopieren eines Ausnahmeobjekts die private Sichtbarkeit des Kopierkonstruktors einfach ignoriert[76]. Ein anderer[77] verlangt zur Compilezeit auch dann einen sichtbaren Kopierkonstruktor, wenn dieser gar nicht aufgerufen wird. Indirektionen innerhalb der Ausnahmeklassen müssen also sehr sorgfältig behandelt oder wenn möglich einfach vermieden werden.
6. Der Destruktor der Klasse `std::exception` ist folgendermaßen definiert:

   ```
   virtual ~exception() throw();
   ```

   Er darf selbstverständlich keine Ausnahme werfen und deklariert das konsequenterweise auch. Der Destruktor Ihrer Ausnahmeklasse sollte auch mit der leeren `throw`-Liste deklariert werden, denn manche Compiler überprüfen dieses Detail und mahnen ein Aufweichen der `throw`-Liste an.

[74] Also kein `std::string`!
[75] Wie zum Beispiel der Borland C++ 5.x-Compiler für Windows.
[76] Der Metrowerks C++ 3.0 für Windows und Mac/PowerPC.
[77] Der GNU gcc 3.4.4. Getestet wurde von mir die Variante cygwin special unter Windows.

7. Verschwenden Sie nicht zu viel Zeit damit, den Text der Methode `what` zu perfektionieren. Vor allem werten Sie diesen Text nicht zur Laufzeit aus! Es können weitere Methoden in die Ausnahmeklasse aufgenommen werden, die alle relevanten Informationen über eine Ausnahme und deren Kontext bereitstellen.
8. In dem Fall, dass Sie Indirektionen in Ihrer Ausnahmeklasse haben (Zeiger, Handles usw.): Beachten Sie bei der Definition des Destruktors die Aspekte, die schon für den Kopierkonstruktor angesprochen wurden. Ausnahmeobjekte können in bestimmten Fällen kopiert werden. Es werden also auch zwei Objekte zerstört. Manche Compiler kopieren Ausnahmeobjekte beim Werfen immer! Um Zeiger abzusichern und den Destruktor der Ausnahmeklasse zu entlasten kann beispielsweise der Autopointer (`std::auto_ptr`) verwendet werden. Dabei ist aber wieder auf den Kopierkonstruktor zu achten. Dieser muss in seiner Initialisierungsliste die Kopierkonstruktoren der `std::auto_ptr`-Objekte aufrufen.

3 Die Objektorientierte Programmierung mit C++

Die Syntax von C++ ist recht aufwändig und im Vergleich zu anderen Sprachen, wie zum Beispiel Java, auch schwerer zu erlernen. Die größere Komplexität von C++ gegenüber anderen objektorientierten Sprachen findet ihre Berechtigung darin, dass mit C++ auch viel systemnaher und im Zweifelsfall auch flexibler programmiert werden kann. Neben der Komplexität ist es ein Nachteil von C++, dass auch viel mehr vom Entwickler selbst erarbeitet werden muss und er nicht auf besondere Eigenschaften einer speziellen Laufzeitumgebung zurückgreifen kann. Die Wahl von C++ für ein Projekt erfolgt meistens aus der Intention, schnelle, kleine oder auch portable Codes zu erzeugen. Dafür nimmt man die Komplexität in Kauf, ja man braucht diese sogar, um gegebenenfalls bezüglich der drei genannten Aspekte Optimierungen durchführen zu können.

Der Syntax liegt aber ein bestimmtes Konzept zugrunde, das in manchen Projekten etwas von der Komplexität der Sprache verdeckt zu werden scheint. Eigentlich sind es zwei sehr unterschiedliche Konzepte, die sich da in C++ vereinen, deren Grundzüge aber nicht durch C++-Syntax beschrieben werden können. Das erste Konzept ist die Objektorientierung und soll in diesem Abschnitt beschrieben werden. Das zweite Konzept der generischen Programmierung wird im Kapitel 4 ab Seite 221 erläutert.

Die Objektorientierung ist ein schon etwa 20-jähriges Konzept zur Modellierung und Beschreibung von Software. Selbst mit Programmiersprachen, die dem Konzept der Strukturierten Programmierung entsprechen, kann es angewandt werden, denn es widerspricht dieser nicht. Die Objektorientierung erweitert die Strukturierte Programmierung und geht weit über sie hinaus. Die Objektorientierung stellt ein höheres Abstraktionsniveau dar als das vorausgegangene Konzept der Strukturierten Programmierung. In ihr werden Prinzipien aus der Linguistik zu Konstruktionszwecken nutzbar gemacht. Auf diesen Unterschied zur Strukturierten Programmierung soll hier näher eingegangen werden.

Programme „laufen ab“. Sie haben einen Anfangs- und einen Endpunkt und Schritte dazwischen[1]. Die Strukturierte Programmierung macht diese Ablauflogik zu ihrem Inhalt, die Objektorientierung arbeitet mit Begriffen, die sich ganz von diesem Gedanken der Befehlssequenz verabschieden und ein Eigenleben entfalten. Während die Strukturierte Programmierung also die „Funktion“ bzw. die „Prozedur“ zur wesentlichen denkerischen Einheit macht, ist diese denkerische Einheit in der Objektorientierung das „Objekt“. Doch was ist ein Objekt? Ein Objekt in den natürlichen Sprachen ist etwas,

[1] Der Einfachheit halber wird die parallele Programmausführung bewusst ausgelassen.

das bezeichnet werden kann. Zum Bezeichnen von Objekten benutzen wir Substantive. Dabei fassen wir unterschiedliche Objekte mit bestimmten Substantiven zusammen. Wenn wir das Substantiv „Tisch" verwenden, können wir damit unterschiedliche Dinge bezeichnen, die aber alle die Eigenschaften haben, die wir in dem Begriff „Tisch" klassifizieren. Es wird mit den Begriffen der Sprache also klassifiziert. Den Begriff als solchen können wir auch als „Klasse" bezeichnen.

Diese Gedankenkette lässt sich ganz analog auf die Objektorientierte Programmierung anwenden. Es steht dort nicht mehr die Funktion als Ablaufeinheit im Mittelpunkt der konstruierenden Arbeit, sondern die Klasse. Die Klasse als Begriff dient der Beschreibung. Es wird in objektorientierten Systemen viel mehr beschrieben als an Algorithmen gefeilt. Die Sprache C++ kennt Klassen und Objekte.

3.1 Der Klassenbegriff

Die Klasse stellt einen Bauplan für ein konkretes Objekt dar. Dabei wird mit der Klasse auch die Gesamtheit der Objekte eines bestimmten Typs bezeichnet. Indem man eine Klasse modelliert, modelliert man das Verhalten aller Instanzen dieser Klasse. Man bestimmt also das Verhalten der Objekte. Nach welchen Gesichtspunkten modelliert man aber eine Klasse? Die Klassen sind also die Beschreibungen der beschreibbaren Dinge im Kontext einer Software. Da sie modelliert werden, handelt es sich um künstliche, also technische Systembeschreibungen. Mit den Klassen werden technische Systeme beschrieben. Was sind eigentlich Systeme? Dazu zunächst einige Definitionen:

Eine Definition aus wikipedia.org

„Klasse ist in der objektorientierten Programmierung ein abstrakter Oberbegriff für die Beschreibung der gemeinsamen Struktur und des gemeinsamen Verhaltens von Objekten (Klassifizierung). Sie dient dazu Objekte zu abstrahieren. Im Zusammenspiel mit anderen Klassen ermöglichen sie die Modellierung eines abgegrenzten Systems."

Definition in Anlehnung an den Systembegriff von N. Luhmann

N. Luhmann[2] prägte als Grundlage seiner Systemtheorie einen einfachen Systembegriff. Er definierte ein System als Grenzziehung zwischen einer Innen- und einer Außenwelt. Die in der Systeminnenwelt vorhandenen Subsysteme sind ebenfalls solche Grenzziehungen. Den Grund der Systembildung nennt er

[2] Niklas Luhmann: * 8. Dezember 1927 in Lüneburg; † 6. November 1998 in Oerlinghausen bei Bielefeld

die Reduktion von Komplexität. Eine Klasse stellt ein System dar. Als System definiert sie eine Grenzziehung zwischen Systeminnenwelt und Außenwelt. Systeme haben die Aufgabe, Komplexität zu reduzieren. Reduziert ein System die Komplexität nicht, so besteht es nicht gegenüber anderen, die diese Eigenschaft haben. Klassen haben also die Aufgabe, durch ihren Systemcharakter Komplexität zu reduzieren.

Definition in Anlehnung an die Linguistik

Eine Klasse repräsentiert einen sprachlichen Begriff, den sich der Softwareentwickler bei der Konstruktion von Software macht. Begriffe sind die Grundbestandteile der Realitäten, die sich der Mensch konstruiert[3]. Bei der Softwareentwicklung handelt es sich um die Konstruktion technischer Realitäten. Er muss die Grundbausteine seiner technischen Vorstellungen identifizieren, um diese Wirklichkeit werden zu lassen. Die Klassen stehen dabei für die Begriffe, die man sich in einem solchen technischen Umfeld macht.

Technische Systeme

Technische Systeme haben die Eigenschaft sich mit der Zeit zu vereinfachen. Die Entwicklertätigkeit, technische Systeme zu schaffen, geht den Weg über die Komplexität zur Einfachheit. Die Schritte, die zur Vereinfachung führen, sind Entkopplung und Modularisierung. Gesamtsysteme werden in handhabbare Subsysteme zerteilt. Bei diesen Subsystemen strebt man dann nach maximaler Entkopplung. Wenn diese Schritte gelungen sind, sind technische Systeme gut zu verstehen, zu bedienen und zu warten.

Genau diese Schritte müssen bei der Entwicklung objektorientierter Software gegangen werden, wenn man beherrschbare Systeme entwerfen möchte. Die Grenzziehung wird durch die Klassenbildung unterstützt. Die Entkopplung muss durch unterschiedliche Mechanismen geleistet werden.

Der Objektbegriff

Der Begriff des Objekts wird in der Objektorientierung etwas unterschiedlich gebraucht. Zum einen meint man mit dem Objekt etwas, das man mit einem sprachlichen Substantiv belegen kann. Keineswegs sind dies nur Dinge, die wir in unserer Umwelt direkt wahrnehmen können, wie es manchmal etwas vereinfachend dargestellt wird. Es sind die Dinge, die wir sprachlich wahrnehmen können. Mit einem Objekt kann beispielsweise ein Buch gemeint sein, aber auch ein Bankkonto. Das erste ist sinnlich erfahrbar, das zweite nicht.

[3] In diesem Zusammenhang möchte ich klarstellen, dass jede Realität konstruiert ist. Sie setzt sich zusammen aus selektiv wahrgenommenem und gedanklich erfassten Bestandteilen. Auch beim gedanklichen Erfassen der Wahrnehmung ist der Mensch auf seine Fähigkeiten der Kategorisierung beschränkt. Mit der Kategorisierung des Wahrgenommenen stellt der Mensch „Bedeutung“ her, die er vernetzt. Damit ist die Realität des Menschen individuell oder im Kommunikationsprozess mit anderen sozial konstruiert.

Obwohl man kaum bestreiten wird, dass der Zustand eines Bankkontos einen durchaus „fühlbaren" Einfluss auf das Leben eines Menschen haben kann. Genauso wie uns häufig Dinge beeinflussen, die nicht dinglich sind, arbeitet der Programmierer mit Objekten, die nichts wiederspiegeln, was direkt dinglich fassbar wäre.

Die Objekte der beschreibbaren Welt haben einen statischen und einen funktionalen Charakter. Die statischen Eigenschaften der Objekte lassen sich durch Attribute beschreiben – in der natürlichen Sprache ebenso wie in C++. Die funktionale Seite ist schwerer zu fassen, da sie sehr vom Kontext abhängen kann, in dem das Objekt beschrieben wird. In einem bestimmten Kontext kann ein Objekt durch ganz andere funktionale Aspekte determiniert sein als in einem anderen. Eine Beschreibung ist immer an einen Kontext gebunden und durch unterschiedliche Aspekte bestimmt. Eine solche Beschreibung eines Objekts nennt man Klasse.

3.2 Die Rolle von Patterns und Idiomen

Die Klasse und das Objekt stellen gewissermaßen die unterste Ebene einer systemischen Betrachtung eines Modells dar. Indem man Klassen identifiziert, kann man Kommunikation definieren, denn die Objekte sind die Teilnehmer von Kommunikationsbeziehungen. Die Kommunikationsbeziehungen zu entwerfen und in einen sinnvollen Zusammenhang zu bringen ist um ein Vielfaches schwieriger, als die reine Identifikation von Klassen. Häufig werden Klassen auch erst durch die Erfordernisse einer bestimmten Kommunikationsform gefunden.

Die Wichtigkeit der Kommunikation zwischen den Objekten, den Komponenten und den Programmteilen im Allgemeinen bringt ein Problem mit sich. Eine objektorientierte Sprache wie C++ versetzt den Entwickler in die Lage über Klassen zu sprechen. Schließlich modelliert der Entwickler Klassen mit C++ und benennt diese mit einem eindeutigen Bezeichner. Dagegen hat C++ keine strukturellen Mittel, die in die Lage versetzten, über die Kommunikation zwischen den Elementen eines Programms zu sprechen. Dabei ist eine Verständigung über die Kommunikation in einer Software von essentieller Wichtigkeit, wenn ein Team gemeinsam entwickelt.

Das Problem, dass für die eigentlich wichtige Kommunikation zwischen den Klassen und Objekten kein Begriffssystem existierte, war eines der Hauptprobleme der Objektorientierten Programmierung der frühen neunziger Jahre. Objektorientierte Sprachen, allen voran C++, setzten sich langsam in der Breite durch, aber es fehlte an eindeutigen Begriffen, die es den Entwicklern erlaubten, über ihre Entwürfe zu sprechen. Mitte der Neunziger erschien das Buch „Entwurfsmuster. Elemente wiederverwendbarer Objektorientier-

ter Software“[4] [GoF95] von den Autoren Gamma, Rumbough, Vlissides und Helms. Dieses Buch revolutionierte die objektorientierte Softwareentwicklung. Es enthält 23 sogenannte Musterbeschreibungen. Muster sind Problem-Lösungspaare, die mit einem Begriff versehen wurden. Dabei erhebt das Buch keinen Anspruch auf Vollständigkeit. Die beschriebenen Lösungen wurden gut durchdacht, mit Angabe von Vor- und Nachteilen einfach enzyklopädisch aneinander gereiht. Für alle Muster wurden Namen gewählt, die für die Beschreibung eines Softwareentwurfs gebraucht werden können[5]. Genau das ist auch die eigentliche Leistung der Publikation. Sie schaffte ein Begriffssystem für den Softwareentwurf.

Heute haben sich die Musterbegriffe für den Softwareentwurf eingebürgert. Einen objektorientierten Softwareentwurf ganz ohne Musterbegriffe beschreiben zu wollen, ist kaum mehr denkbar. Allen Musterbeschreibungen fügten die Autoren eine Diskussion über Vor- und Nachteile und Ausprägungen an. Ein Muster ist also nicht einfach eine Idealimplementierung einer Lösung für ein Problem. Ein Muster ist immer eine ganze Schar möglicher Lösungen; unterschiedliche Implementierungen, die alle einen ähnlichen Grundzug aufweisen.

Für die moderne C++-Entwicklung ist die Kenntnis der wesentlichen Muster eine wichtige Voraussetzung für erfolgreiche Teamarbeit. Die beschriebenen Muster in [GoF95] sind:

Abstrakte Fabrik	Erbauer
Fabrikmethode	Prototyp
Singleton	Adapter
Brücke	Dekorierer
Fassade	Fliegengewicht
Kompositum	Proxy
Befehl	Beobachter
Besucher	Interpreter
Iterator	Memento
Schablonenmethode	Strategie
Vermittler	Zustand
Zuständigkeitskette	

Davon sind nicht alle in der gleichen Weise wichtig. Einige aber sind fast schon zentral für die Arbeit am Softwareentwurf mit C++. Es wird hier deshalb ein Überblick über drei der oben genannten Muster gegeben.

[4] Der Originaltitel in Englisch lautete „Design Patterns. Elements of Reusable Object-Oriented Software“.

[5] Das Buch stieß in eine echte Lücke und wurde milionenfach verkauft.

3.2.1 Der Iterator

Seit dem Anfang der Neunziger Jahre werden mit den C++-Entwicklungssystemen Bibliotheken ausgeliefert, die einfache Grundcontainer wie verkettete Listen oder dynamische Vektoren enthalten. Diese Container waren zwar herstellerspezifisch, jedoch kam meistens ein einheitliches Konzept zur Anwendung, wie auf die Elemente der Container zugegriffen werden konnte. Dieses Konzept besteht darin, den Zugriff auf den Container als solchen in einem extra dafür geschaffenen Typ zu kapseln und damit von der eigentlichen Datenhaltung zu separieren. Spätestens mit der Entwicklung der Standard Template Library durch Alexander Stephanov[6] setzte sich dieses Prinzip in der Containerimplementierung allgemein durch. Denn es bietet viele Vorteile: Zum einen stellt diese „Iterator"-Schnittstelle eine Vereinheitlichung des Zugriffs auf unterschiedliche Containertypen dar, so dass der Entwickler sich weniger um die interne Struktur der Datenhaltung kümmern muss. Zum anderen vereinfacht es auch die Implementierung der Datencontainer selbst beträchtlich, denn es müssen in den Containern keine Zustandsinformationen der Traversion gespeichert werden.

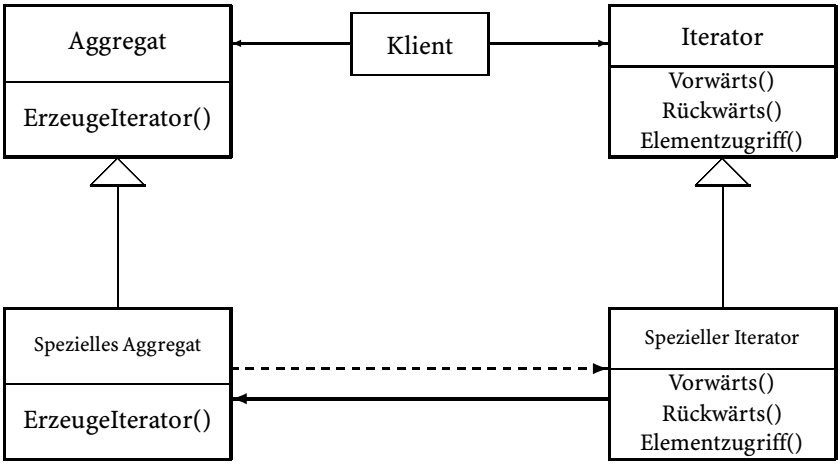


Abbildung 3.1. Der Iterator

In der Abbildung wird das Klassendiagramm des Iterator-Musters nach Gamma et al. gezeigt. Dieser Implementierungsvorschlag beruht auf klassischen objektorientierten Techniken. Er nutzt den Polymorphismus mit den

[6] Siehe Abschnitt 5.2.6 auf Seite 292.

spät gebundenen Methoden. Sowohl die möglichen Aggregate wie auch die dazu passenden Iteratoren sind polymorph. Diese Implementierungsweise findet man vorwiegend in älteren C++-Bibliotheken und -Frameworks. Man findet sie auch in hochgradig objektorientierten Systemen, wie zum Beispiel in den Java-Bibliotheken. Mit der Implementierung der STL auf Basis der Templatetechniken wurden andere Wege zur Realisierung von Iteratoren eingeschlagen. Man definierte die Schnittstelle des Iterators nicht dadurch identisch, dass man ihn von einer abstrakten Basisklasse ableitete, sondern indem man durch Templates garantierte, dass alle Methoden immer in der gleichen Weise benannt wurden. Dabei unterscheiden sich aber die Rückgabetypen einiger Methoden und passen sich damit den Elementtypen der Container an. Der letzte Punkt ist ein Vorteil, der durch die objektorientierte Variante nicht zu haben ist ohne beim Datenzugriff auf Typenkonvertierungen zurückgreifen zu müssen. Wie die Iteratoren in der STL funktionieren, wird in Abschnitt 5.2.6 ab Seite 308 beschrieben.

Das beschriebene Iterator-Muster steht einfach für die Idee, die Traversierung über eine dynamische Datenstruktur und den Zugriff auf Elemente in der Struktur in einen separaten Typ auszulagern. In vielen Fällen ist es ein Klassentyp, wie in dem Strukturvorschlag von [GoF95] beschrieben. Die Beachtung dieser Modellierungsidee führt zu einfacheren Implementierungen der dynamischen Datenstrukturen und zu einer besseren Anwendbarkeit derselben.

3.2.2 Das Zustandsmuster

In der UML gibt es zur Modellierung von Zuständen eine extra Diagrammform: das Zustandsdiagramm. Man hat dem Zustandsgedanken dieses Diagramm gewidmet, da es sich als sehr hilfreich erwiesen hat, bei der Modellierung komplexer Verhaltensweisen in Zuständen zu denken[7]. Viele technische Sachverhalte lassen sich mittels eines Zustandsdiagramms beschreiben. Dabei hat ein solches Zustandsdiagramm die Besonderheit, dass man mit einem Regelsatz daraus Code erzeugen kann. In diesem Punkt hat das Zustandsdiagramm sogar dem Klassendiagramm etwas voraus. Ein Klassendiagramm ist im Allgemeinen reduktionistisch. Das heißt, es reduziert den Sachverhalt so, dass etwas Wesentliches besser darstellbar und kommunizierbar wird. Ein Klassendiagramm sollte daher auch nicht nach „Vollständigkeit“ in dem Sinne streben, dass es alle möglichen Elemente einer Klasse sichtbar macht. Damit wird die eigentliche Designidee eines Klassendiagramms nicht mehr lesbar. Ein Zustandsdiagramm enthält eine Designidee und kann trotzdem

[7] In diesem Zusammenhang ist mit „Zustand“ nicht irgendein Schnitt in einem kontinuierlichen Werteverlauf gemeint, sondern eine abgrenzbare Existenzform. Die Anzahl beschreibbarer Zustände in diesem Sinne ist also endlich.

eine gewisse Vollständigkeit erlangen. Die Designidee für ein Zustandsdiagramm kann zum Beispiel in der Hierarchisierung der beschriebenen Zustände bestehen. Eine weitere Leistung des Zustandsdiagramms kann es sein, dass Zustände separiert voneinander dargestellt werden können. Mehrere Zustandsabläufe können unabhängig voneinander modelliert werden oder eben synchronisiert werden. Des Weiteren können die Nachrichten eingezeichnet werden, die dazu führen, dass Zustände gewechselt werden.

Ein sehr wichtiges Detail für die Modellierung von Zuständen ist deren Kontext. Erst wenn man den Kontext genau kennt, kann man überhaupt Zustände identifizieren. Ungenauigkeiten im Zusammenhang mit der Bestimmung des Kontextes führen oft zu unsauberen und schwer wartbaren Zustandsmaschinen. Deshalb machen Gamma et al. in ihrer Publikation [GoF95] den Kontext sichtbar. Der Vorschlag, der durch die Autoren des Buchs zur Implementierung von Zuständen geliefert wurde, basiert auf einfachem Polymorphismus.

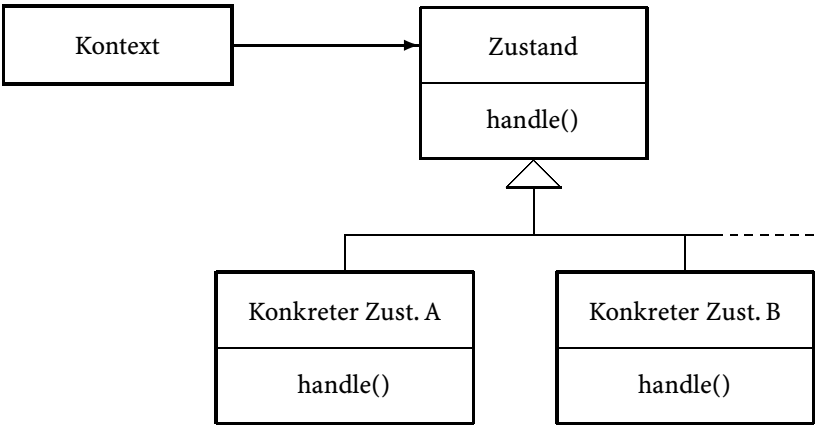


Abbildung 3.2. Das Zustandsmuster

Der Zustandsbegriff wird in technischen Beschreibungen verwendet, um unterschiedliches Verhalten in einem Kontext X zu bezeichnen. Meistens fehlen für den Beschreibungsgegenstand andere Begrifflichkeiten, weshalb man sich des Zustandsbegriffs bedient. Die breite Anwendbarkeit verdankt dieser Begriff seiner Abstraktionsebene. Der Zustandsbegriff als solcher ist sehr abstrakt und daher einfach anzuwenden.

Ist eine Zustandsbeschreibung einmal fertig, kann sie in die Implementierung überführt werden. Das Diagramm des Zustandsmusters aus [GoF95] weist den Weg, wie die Implementierung aussehen könnte. Für die Zustände,

die in direktem Zusammenhang stehen – die also ineinander übergehen können –, müssen Basisklassen gefunden werden. Schließlich werden die konkreten Zustände auch als Klassen modelliert und leiten von diesen Basisklassen ab. Die Basisklassen sind manchmal nicht ganz leicht zu benennen. Denkerisch lassen sie sich meist leicht erfassen, es fehlt oft an geeigneten Bezeichnungen. Für die Implementierung ist nach [GoF95] die Methode `handle()` wichtig. Diese symbolisiert nur eine oder mehrere virtuelle Methoden der Zustandsklassen. Wie diese heißen, ist also egal. Wichtig ist nur, dass diese Methoden das eigentliche Verhalten der Zustände definieren.

Ebenso wichtig sind die Methoden, die den Zeitpunkt des Zustandseintritts und des Zustandsaustritts kennzeichnen. Man könnte sie etwas allgemein mit `enterState()` und `exitState()` bezeichnen. Auch diese sind virtuell und legen Verhalten fest. Sie stehen für ganz spezielle Zeitpunkte, die es bei der Arbeit mit Zuständen zu beachten gilt. Denkt man an die drei Arten von Methoden, die mit Zuständen im Zusammenhang stehen, dann hat man die wesentlichen Stellen im Design bereits identifiziert, an denen Zustandsverhalten modifiziert werden kann.

Ein weiterer wichtiger Aspekt im Zusammenhang mit Zustandsmaschinen ist die Beschreibung des Kontextes. Der Kontext definiert die Zustandsreihenfolge. Das macht er entweder in seiner Initialisierung oder er macht es später beim Schalten der Zustandswechsel. Ob man es in der Initialisierung macht oder an anderer Stelle algorithmisch löst, ist im Allgemeinen nicht so wichtig. Der Kontext ist der Ort, in dem man die Zustandsreihenfolge modellieren kann. Im nächsten Abschnitt soll eine einfache Beispielimplementierung demonstrieren, wie die angesprochenen Elemente ganz regelhaft für eine Implementierung genutzt werden können.

Ein einfaches Beispiel: die Ampel

Ein typisches Beispiel für eine Zustandsmaschine, einen Kontext also, in dem Zustände ablaufen, ist die Verkehrsampel. Im Folgenden soll eine einfache Verkehrsampel modelliert werden[8]. Dafür kann ein Zustandsdiagramm entworfen werden.

Zunächst kann man die vier Ampelphasen voneinander unterscheiden und in einen Zustandskreislauf bringen. Die Ampelphasen lassen sich zu Recht als Zustände auffassen. Ihr „Verhalten“ unterscheidet sich in ihren Bedeutungen, die sie für die Autofahrer haben und auch darin, welchen Folgezustand sie definieren. Wenn man bedenkt, dass eine Ampel auch gelb blinken kann und damit zum Ausdruck bringt, dass sie den Verkehr zu diesem Zeitpunkt nicht regelt, ist ein weiterer Zustand identifiziert. Dieser blinkende, nicht regelnde Zustand ist nicht ebenbürtig mit einer der vier Phasen

[8] Natürlich ohne jede Verantwortung für deren verkehrstechnische Funktion und Vollständigkeit zu übernehmen. Die Ampel soll nur ein einfaches Beispiel sein für eine Zustandsmaschine. Sie wurde ohne jede Kenntnis gesetzlicher Regelungen zu diesem Thema geschrieben.

vergleichbar. Er steht gewissermaßen außerhalb. Man kann diesem nicht regelnden Zustand einen regelnden Zustand entgegenhalten, in dem die Ampelphasen ablaufen. Im Folgenden werden diese übergeordneten Zustände „Betriebszustände“ genannt. Das ist die begriffliche Hierarchisierung, die mit dem Zustandsdiagramm zum Ausdruck gebracht werden soll. Damit ist der regelnde Betriebszustand `InBetrieb` gleichzeitig der Kontext der Ampelphasen. Der Einfachheit halber ist der nicht regelnde Betriebszustand nicht weiter unterteilt. Für einen realen Ampelaufbau müssen sicher noch ganz andere Aspekte betrachtet werden, die in die Modellierung der Zustandsmaschine mit einfließen müssen.

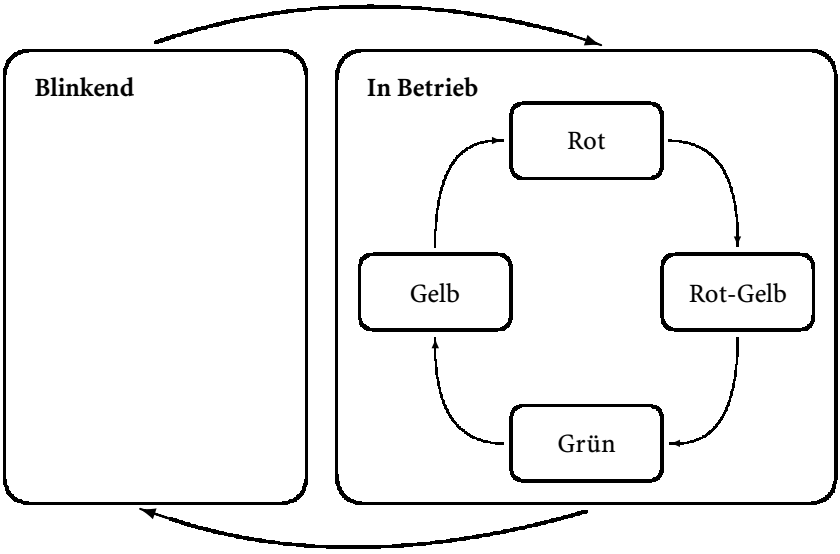


Abbildung 3.3. Das Zustandsdiagramm des Ampelbeispiels

Geht man in der Implementierung nach dem Vorschlag von [GoF95], dann müssen Kontext und Zustände als Klassen modelliert werden. Wie schon erwähnt wurde, ist dabei der Kontext für die Zustandsreihenfolge verantwortlich. In den Zuständen selbst sind drei Zeitpunkte für die Modellierung wichtig. Der Zustandseintritt, die zeitliche Phase des Bestehens und der Zustandsaustritt. Diese Zeitphasen werden durch Methoden wie `enterState()`, `exitState()` und `handleEvent()` abgedeckt. Im folgenden Beispiel enthalten die Zustandsklassen auch einen Zeiger auf die Folgezustände. Initialisiert werden diese Zeiger aber durch den Kontext, der aus seiner Verantwortung für die Zustandsreihenfolge nicht entlassen wird. Für die Betriebszustände ist der Einfachheit halber die Ampel der Kontext.

Listing 3.1. Implementierung des Ampelbeispiels

```
#include <iostream>
using namespace std;

typedef char EVENT;

class Phase
{
public:
  virtual ~Phase();

  virtual void enterState() = 0;
  virtual void exitState() = 0;
  virtual void handleEvent( EVENT ) = 0;

  Phase *folgezustand;
};

class Rot : public Phase
{
public:
  void enterState();
  void exitState();
  void handleEvent( EVENT );
};

class Gelb : public Phase
{
public:
  void enterState();
  void exitState();
  void handleEvent( EVENT );
};

class Gruen : public Phase
{
public:
  void enterState();
  void exitState();
  void handleEvent( EVENT );
};
```

```
class RotGelb : public Phase
{
public:
  void enterState();
  void exitState();
  void handleEvent( EVENT );
};

class Betriebszustand
{
public:
  virtual ~Betriebszustand();

  virtual void enterState() = 0;
  virtual void exitState() = 0;
  virtual void handleEvent( EVENT ) = 0;

  Betriebszustand *folgezustand;
};

class InBetrieb : public Betriebszustand
{
public:
  InBetrieb();
  void enterState();
  void exitState();
  void handleEvent( EVENT );

private:
  Rot rot;
  Gelb gelb;
  RotGelb rotgelb;
  Gruen gruen;
  Phase *aktuellerzustand;
};

class Blinkend : public Betriebszustand
{
public:
  void enterState();
  void exitState();
  void handleEvent( EVENT );
};
```

```
class Ampel
{
public:
  Ampel();

  bool handleEvent( EVENT e );

private:
  InBetrieb inbetrieb;
  Blinkend blinkend;
  Betriebszustand *aktuellerzustand;
};

//////////////////////////////////////

Ampel::Ampel()
{
  inbetrieb.folgezustand = &blinkend;
  blinkend.folgezustand = &inbetrieb;
  aktuellerzustand = &inbetrieb;
}

bool Ampel::handleEvent( EVENT e )
{
  bool ret = true;
  switch( toupper( e ) )
  {
  case 'B':
    aktuellerzustand->exitState();
    aktuellerzustand = aktuellerzustand->folgezustand;
    aktuellerzustand->enterState();
    break;
  case 'X':
    ret = false;
  default:
    aktuellerzustand->handleEvent( e );
  }
  return ret;
}

Phase::~Phase() {}

void Rot::enterState() { cout << "ROT" << endl; }
```

```
void Rot::exitState() {}
void Rot::handleEvent( EVENT ) {}

void RotGelb::enterState()
{ cout << "ROT-GELB" << endl; }
void RotGelb::exitState() {}
void RotGelb::handleEvent( EVENT ) {}

void Gelb::enterState() { cout << "GELB" << endl; }
void Gelb::exitState() {}
void Gelb::handleEvent( EVENT ) {}

void Gruen::enterState() { cout << "GRUEN" << endl; }
void Gruen::exitState() {}
void Gruen::handleEvent( EVENT ) {}

Betriebszustand::~Betriebszustand() {}

InBetrieb::InBetrieb()
{
  rot.folgezustand = &rotgelb;
  rotgelb.folgezustand = &gruen;
  gruen.folgezustand = &gelb;
  gelb.folgezustand = &rot;
  aktuellerzustand = &rot;
}
void InBetrieb::enterState()
{
  aktuellerzustand = &rot;
  aktuellerzustand->enterState();
}
void InBetrieb::exitState()
{
  aktuellerzustand->exitState();
}
void InBetrieb::handleEvent( EVENT e )
{
  switch( toupper( e ) )
  {
  case 'F':
    aktuellerzustand->exitState();
    aktuellerzustand = aktuellerzustand->folgezustand;
    aktuellerzustand->enterState();
```

```
    break;
  default:
    aktuellerzustand->handleEvent( e );
  }
}

void Blinkend::enterState()
{ cout << "blinkend" << endl; }
void Blinkend::exitState() {}
void Blinkend::handleEvent( EVENT ) {}

int main()
{
  Ampel ampel;

  EVENT e;
  do
  {
    cin >> e;
  } while( ampel.handleEvent( e ) );

  return 0;
}
```

In dem Beispiel in Listing 3.1 sind viele leere Methoden enthalten. Sie repräsentieren die Stellen, in denen das Verhalten der State-Machine definiert werden kann.

3.2.3 Das Singleton - Muster

Die Überlegungen, die im Folgenden zum Singleton-Muster angestellt werden, wurden von mir im Wesentlichen schon 2003 in Artikelform im Web veröffentlicht. Es liegt diesem Abschnitt also der Artikel zugrunde, der unter `http://www.oop-trainer.de/Themen/Singleton.html` zu finden ist. Außerdem sind die Reaktionen auf den Artikel aus dem Web eingearbeitet. Die Urheber und die Adressen im Netz werden dabei genannt.

In dem Buch „Design Patterns. Elements of Reusable Object-Oriented Software“ [GoF95] haben die Autoren Erich Gamma, Richard Helm, John Vlissides und Ralph Johnson ein strukturell sehr einfaches und inzwischen weit bekanntes Muster beschrieben: Das Singleton. Die Implementierungssprache, die für die Beispiele gewählt wurde, ist C++. Das legt natürlich die Verwendung dieses Musters in der genannten Sprache nahe, auch wenn gerade der Einsatz des Singletons in C++ von einigen strukturellen Problemen

begleitet wird, wie in diesem Abschnitt zu zeigen sein wird. Im Gegensatz zu einem Einsatz des Singletons in Java müssen Rahmenbedingungen genauer geprüft und Ziele exakter definiert werden, wenn man das Muster in C++ gewinnbringend verwenden möchte. Zunächst kann man natürlich versucht sein, den Beispielcode aus [GoF95] für eigene Projekte nutzbar zu machen. Dazu sollte man diesen allerdings etwas genauer unter die Lupe nehmen.

Listing 3.2. Singletonimplementierung nach [GoF95]

```
class Singleton
{
   public:
     static Singleton* exemplar();

   protected:
     Singleton() {}

   private:
     static Singleton *instanz;
};

Singleton* Singleton::instanz = 0;

Singleton* Singleton::exemplar()
{
  if( instanz == 0 )
    instanz = new Singleton();
  return instanz;
}
```

In Listing 3.2 steht eine dem Buchbeispiel analoge Implementierung. Der Sinn des Singletonmusters besteht in erster Linie darin, nur eine Instanz einer Klasse zuzulassen. Prüfen wir die Implementierung auf Tauglichkeit vor dieser Anforderung, so fallen Lücken auf. Zum einen hat der Standardkonstruktor eine protected-Sichtbarkeit, was dazu führen kann, dass man durch einfaches Ableiten die Sichtbarkeit des Konstruktors umbeabsichtigt öffnet. Zum anderen wurde der Kopierkonstruktor ganz vergessen. Das heißt, dass der Compiler automatisch einen solchen generiert – mit der Sichtbarkeit public. Mit dem Kopierkonstruktor kann man also eine weitere Instanz erzeugen, ohne dass man durch die Struktur des Buchbeispiels daran gehindert werden würde. Will man also die Klasse gemäß der Idee des Singletons schützen, so sind Modifikationen nötig. Eine weitere Auffälligkit ist die Verwendung des new-Operators. Dieser wird gebraucht, um die Objektinstanz dynamisch auf dem Heap anzulegen. Andernfalls könnte man eine Instanz auch als statisches Objekt anlegen. In vielen Fällen wird das auch die sinnvollere Vorgehenswei-

se darstellen. Die dynamische Instanziierung ergibt vor allem dadurch Sinn, dass nicht in jedem Fall instanziiert werden muss. Wenn das Objekt sehr groß ist oder Ressourcen gebraucht werden, die nicht in jedem Programmlauf benötigt werden, ist die Instanziierung durch new einer statischen vorzuziehen. Dabei stellt sich aber die Frage nach dem Löschen des Objektes. In dem vorgebrachten Beispiel ist kein struktureller Bestandteil enthalten, der sich um das Beseitigen der Instanz kümmert. Wer nun nicht darauf vertrauen möchte, dass das Betriebssystem den Speicher schon wegräumen werde, und wer einen Destruktoraufruf der Objektinstanz braucht, um andere Ressourcen freizugeben, der wird sich eine Lösung überlegen müssen. Die Lösung bedeutet eine strukturelle Änderung, wenn sie systemimmanent sein soll. Nach diesen Vorüberlegungen zu dem von [GoF95] gegebenen Implementierungsbeispiel ist es nun möglich, erste Verbesserungsvorschläge zu erarbeiten, zunächst ohne dabei auf besondere Ausprägungen des Musters einzugehen, wie sie beispielsweise in der Publikation von J. Vlissides diskutiert werden (dort wird das Singleton im Zusammenhang mit Referenzzählung betrachtet und fortentwickelt).

Listing 3.3. Singleton mit privatem Kopierkonstruktor

```
class Singleton
{
   public:
     static Singleton* exemplar();

   protected:
     Singleton() {}

   private:
     static Singleton *instanz;
     Singleton( const Singleton& );
};

...
```

Das am einfachsten zu lösende Problem ist das mit dem Kopierkonstruktor: man muss ihn einfach nur mit privater Sichtbarkeit deklarieren. Privat deklariert ist der Kopierkonstruktor nicht mehr vom Compiler generierbar und kann auch nicht mehr aufgerufen werden. Eine Instanz lässt sich durch ihn nicht mehr erstellen. Etwas komplexer verhält es sich mit der Sichtbarkeit des Standardkonstruktors: die protected-Sichtbarkeit, wie sie bei [GoF95] angegeben ist, deutet auf eine gewollte Vererbbarkeit der Singletonklasse hin. Damit muss auch die Singletoneigenschaft vererbbar sein. Betrachtet man die aus einer Vererbung resultierenden Kindklassen, so stellt man fest, dass die statischen Elemente der Singletonelternklasse nur einmal für alle abgeleiteten

Klassen gemeinsam existieren. Damit wird also nicht ein Verhalten realisiert, das gewährleistet, dass die Kindklassen jeweils nur einmal instanziiert werden können, sondern ein Verhalten, das nur eine Instanz irgend einer Kindklasse erlaubt. Das ist ein ziemlich bemerkenswertes Verhalten, das, wenn es gewünscht wird, in der vorliegenden Beispielimplementierung mit dem geschützten Standardkonstruktor fast gelöst ist. Um das Problem vollständig zu lösen, muss für eine solche Kindklasse noch die Instanziierungsmethode neu geschrieben werden und der Standardkonstruktor versteckt werden (und schon findet man fast die ganze Singletonstruktur in der Kindklasse wieder). Der Sinn einer solchen Elternklasse mit dem angesprochenen Verhalten ist also äußerst zweifelhaft. Meint man jedoch, mit der Vererbung Kindklassen erzeugen zu können, die eine proprietäre Singletoneigenschaft aufweisen, so taugt die Beispielimplementierung nicht. Es muss also gefragt werden, ob für eine Singletonimplementierung die angesprochene Eigenschaft überhaupt wünschenswert ist. Wenn nicht – was in den meisten Fällen zutrifft –, dann sollte der Konstruktor die private Sichtbarkeit bekommen, um eine Vererbung unmöglich zu machen, und es muss der fachliche Code mit dem Mustercode gemischt werden. Diese nicht sehr elegante Variante ist zumindest sicher und garantiert die Singletoneigenschaft ohne sie durch eine mögliche Vererbung aufzubrechen.

Listing 3.4. Singleton mit privatem Standardkonstruktor

```
class Singleton
{
   public:
     static Singleton* exemplar();

     void fachlicheFunktion1();
     void fachlicheFunktion2();
     ...

   private:
     static Singleton *instanz;
     Singleton() {}
     Singleton( const Singleton& );
};

...
```

Will man allerdings eine allgemeine Klasse definieren, die als Singletonelternklasse die Singletoneigenschaft proprietär auf ihre Kindklassen vererbt, muss man andere Wege gehen, die an einer späteren Stelle in diesem Abschnitt erörtert werden sollen. Zunächst soll auf das Problem des Löschens eingegangen werden: Listing 1 zeigt die statische Methode, die die Instanz der

Klasse erzeugt. Dort wird die Instanz mittels new auf dem Heap angelegt. Alternativ könnte auch eine statische Instanz angelegt werden, deren Referenz – oder deren Zeiger – die Methode zurückgibt. Die Konsequenz ist allerdings, dass schon zur Startzeit des Prozesses der Speicher vorreserviert wird (was in vielen Fällen vertretbar ist). Größere Speichermengen können ja dynamisch im Konstruktor der Klasse angefordert werden. Auch in dieser Implementierung kann der fachliche Code untergemischt werden. Die Instanz wird beim Prozessende ordentlich beseitigt und es wird der Destruktor aufgerufen. Für diesen ist allerdings eine Rahmenbedingung wirksam: der Zeitpunkt des Destruktoraufrufes fällt zusammen mit den Destruktoraufrufen aller anderen globalen Instanzen des Prozesses. Sollte nun ein Design generell so konzipiert sein, dass man möglichst keine globalen Instanzen irgendwelcher Klassen benutzt, so sind doch einige globale Objekte aus der Standardbibliothek vorhanden, die ebenfalls abgebaut werden. In welcher Reihenfolge diese Objekte nun beseitigt werden, ist nicht festgelegt. Aus diesem Grund darf der Destruktor der Singletonklasse auf keine globalen Fremdobjekte mehr zugreifen, denn diese könnten ihrerseits bereits abgeräumt sein. Kann man die Rahmenbedingung erfüllen und bereitet das Vorreservieren des Speichers keine Probleme, so ist diese sicherlich die unproblematischste aller Singletonimplementierungen.

Listing 3.5. Singleton mit statischer Instanz

```
class Singleton
{
   public:
     static Singleton& exemplar();

   private:
     Singleton() {}
     Singleton( const Singleton& );
};

Singleton& Singleton::exemplar()
{
  static Singleton instanz;
  return instanz;
}
```

In dem Fall, wenn die Instanz aus Platzgründen nicht statisch angelegt werden kann und new verwendet werden soll, muss eine andere Möglichkeit zum Abräumen der Singletoninstanz geschaffen werden. Das kann beispielsweise durch ein statisches Wächterobjekt geschehen, das minimalen Platz benötigt und bei Prozessende, im Falle der Instanziierung des Singletons, die Instanz löscht. Damit dieses Objekt an der Stelle gekapselt ist, wo es benö-

tigt wird, kann man dessen Klasse als verschachtelte Klasse private innerhalb des Singletons deklarieren. Damit wird auch das Problem der undefinierten Initialisierungsreihenfolge der globalen Objekte umgangen. Die Singletoninstanz wird, obwohl sie statisch ist, nach den globalen Objekten initialisiert. Auch hier gilt: Der Destruktor der Singletonklasse wird durch den Destruktor der statischen Wächterinstanz aufgerufen, also während der Aufräumarbeiten aller globalen Objektinstanzen. Da die Reihenfolge der Objektlöschungen nicht festgelegt ist, kann auf kein anderes globales Objekt zu diesem Zeitpunkt zugegriffen werden. Die vorgestellte Lösung mit der Wächterinstanz hat gegenüber der Lösung mit der statischen Instanz also nur den Vorteil, dass während der Laufzeit Ressourcen gespart werden können, wenn die Singletoninstanz gerade nicht benötigt wird und bei einer Instanziierung viele Ressourcen verbrauchen würde. In Listing 3.6 finden Sie die Lösung mit der Wächterinstanz. Bei einigen Compilern zieht die Instanziierung der Wächterinstanz, wie sie im Listing dargestellt ist, eine Warnung nach sich, denn es wird ein Bezeichner eingeführt, auf den niemals zugegriffen wird. Dafür kann natürlich durch entsprechende Compilerpragmas oder Warninglevels Abhilfe geschaffen werden. Wenn man eine compilerübergreifende Lösung haben möchte, kann man eine leere Inlinemethode in der Wächterklasse definieren, die nach der Zeile mit der statischen Instanziierung aufgerufen werden kann. Die Compiler sind heute fast alle in der Lage, eine solche leere Inlinemethode wegzuoptimieren. Die Warnung ist damit strukturell beseitigt. Da nun aber das Löschen exklusiv durch das Singleton selbst geschehen soll, sollte man den eigentlichen Destruktor privat machen. Damit wird das Löschen außerhalb des Scopes der Singletonklasse unterbunden.

Listing 3.6. Singleton mit Wächterklasse

```
class Singleton
{
   public:
     static Singleton* exemplar();

   private:
     static Singleton *instanz;
     Singleton() {}
     Singleton( const Singleton& );

     ~Singleton() {}

     class Waechter {
         public: ~Waechter() {
           if( Singleton::instanz != 0 )
             delete Singleton::instanz;
         }
```

```
    };
    friend class Waechter;
};

Singleton* Singleton::instanz = 0;

Singleton* Singleton::exemplar()
{
  static Waechter w;
  if( instanz == 0 )
    instanz = new Singleton();
  return instanz;
}
```

Im letzten Abschnitt wird auf das Problem einzugehen sein, dass die bis jetzt entwickelten Singletonimplementierungen ein proprietäres Vererben der Singletoneigenschaft auf Kindklassen nicht erlauben. Zur Erinnerung: Die statischen Elemente einer Elternklasse werden durch Kindklassen gemeinsam genutzt. Will man erreichen, dass für jede Kindklasse eigene statische Elemente zur Verfügung stehen, so muss für jede dieser Klassen eine exklusive Singletonelternklasse existieren. Wenn man das von Hand erreichen wollte, wäre das ein enormer zusätzlicher Schreibaufwand, der den Einsatz der Musterlösung an sich in Frage stellen würde. Zu Hilfe kommt hier die Möglichkeit in C++, durch Templates den Compiler die Klassen erzeugen zu lassen: man kann eine Elternklasse in Abhängigkeit von der Kindklasse erzeugen lassen. Dabei kann man auch gleich der statischen Instanziierungsmethode den gewünschten Produkttyp einschieben. Realisierbar ist diese Lösung sowohl mit dem Grundgerüst der Implementierung mit der Wächterinstanz als auch mit der auf der statischen Singletoninstanz beruhenden Implementierung. Für eine Demonstration wird hier die letztgenannte gewählt, da sie etwas einfacher ist und mit weniger Zeilen Code auskommt. Die Variante mit Wächterinstanz lässt sich analog für eine Template-Imlementierung verwenden. Allerdings können bei einigen Compilern Probleme mit der Übersetzbarkeit auftreten, wenn diese den ANSI/ISO-C++-Standard nicht korrekt umsetzen. Um diese Probleme im Einzelfall zu umgehen und um die Codegröße auf ein Minimum zu reduzieren, wählte ich für die Beispielimplementierung in Listing 3.7 die Variante mit statischer Instanziierung. Da die Singletonimplementierung nun eine Elternklasse sein soll, die ihre Eigenschaften auf die Kindklassen vererbt, muss der Standardkonstruktor public oder protected sein. Was sich strukturell ändert, ist nur der Einschub der Kindklasse als Produkttyp in die Instanziierungsmethode und die Vervielfältigung dieser Methode. Etwas seltsam mag dem einen oder anderen auch die Verwendung des Kindklassentyps zur Parametrisierung des Elternklassentemplates vorkommen (gerade das ist die problematische Stelle, die bei einer analogen Verwendung der Wächterim-

plementierung zu Compilerfehlern führen kann, da dort stärker auf Typinformationen der Kindklasse Bezug genommen werden muss). Eine solche Verwendung des Kindklassentyps ist jedoch unproblematisch, da Informationen über den inneren Aufbau der Klasse nur in der Instanziieringsmethode gebraucht werden. Diese ist aber unabhängig von einem Objekt der Elternklasse.

Listing 3.7. Template-Implementierung der Elternklasse(n)

```
template <class Derived>
class Singleton
{
   public:
     static Derived& exemplar();

   protected:
     Singleton() {}

   private:
     Singleton( const Singleton& );
};

template <class Derived>
Derived& Singleton<Derived>::exemplar()
{
  static Derived instanz;
  return instanz;
}

Verwendung:

class ChildA : public Singleton<ChildA>
{  ...  };

class ChildB : public Singleton<ChildB>
{  ...  };
```

Ein Problem, das wir mit dieser auf Templates basierenden Implementierung haben, ist, dass die Konstruktoren der Kindklassen nicht mehr verborgen sind. Wenn man sie verbergen möchte, müsste man für die Elternklasse eine friend-Deklaration einführen. Das kann z. B. über ein Makro geschehen oder eben direkt. Das bedeutet aber zusätzlichen Aufwand, der die Eleganz der Lösung in Frage stellt. Die erzeugten Kindklassen haben also alle Elemente eines Singletons mit Ausnahme der verborgenen Konstruktoren. Mit entsprechender Zusatzinformation lassen sie sich aber verwenden (Instanz darf nur über die Instanziierungsmethode angefordert werden).

Zuallerletzt möchte ich auch dafür noch einen Vorschlag machen: die unproblematischen Singletonimplementierungen sind die, die ihre Musterstruktur mit dem fachlichen Code mischen ohne dabei Vererbung einzusetzen. Die Elemente des Musters können also alle mittels eines Makros in eine Klasse eingefügt werden. Das Makro kann die Klassenelemente schon mit der richtigen Sichtbarkeit deklarieren. Für die Definition der statischen Elemente außerhalb der Klassen ist dann ein weiteres Makro zuständig. Das könnte etwa so aussehen, wie es in Listing 3.8 demonstriert wird. Die Makros müssten nur die Deklarationen bzw. die Definitionen einer der beiden Mustervarianten (Wächterinstanz in Listing 3.5 oder statische Singletoninstanz in Listing 3.6) wiederholen. Auf die Implementierung der Makros wird an dieser Stelle verzichtet, da sie mehr oder weniger trivial ist. Der Vorteil dieser Lösung liegt in ihrer einfachen Handhabung. Der Compiler gibt Fehlermeldungen aus, wenn man das implementierende Makro vergisst anzugeben.

Listing 3.8. Verwendung eines Singletons auf Makrobasis

```
class MyClass
{
    DECLARE_SINGLETON(MyClass);

    void fachlicheFunktion1();
    void fachlicheFunktion2();
    ...
};

...
DEFINE_SINGLETON(MyClass);
```

Einen Vorschlag zur Implementierung zur Mischung des Singletoncodes mit dem fachlichen Code liefert Robin Doer auf seiner Seite `www.robind.de`. Er definiert eine Template-Wächterklasse, die durch beliebige Klassen verwendet werden kann, die die Singletoneigenschaft brauchen.

Listing 3.9. Definition eines auf Templates basierenden Wächters

```
template <class T>
class SingletonWaechter {
  T* singleton;

public:
  SingletonWaechter()
  {
    singleton = new T;
  }
  ~SingletonWaechter()
```

```
  {
    if (singleton != 0)
      delete singleton;
  }

  T* instance() const
  {
    return singleton;
  }
};

class CppClass {
public:
  static CppClass* getInstanz();

private:
  CppClass() {}
  CppClass(const CppClass& other);
  ~CppClass() {}

  friend class SingletonWaechter<CppClass>;
};

CppClass* CppClass::getInstanz()
{
  static SingletonWaechter<CppClass> w;
  return w.instance();
}

int main()
{
  CppClass *p = CppClass::getInstanz();

  return 0;
}
```

In diesem Abschnitt wurden weder besondere Ausprägungen des Singletonmusters beschrieben, noch Fragestellungen des Softwaredesigns angesprochen, die für oder gegen die Verwendung von Singletons sprechen. Es sollte nur aufgezeigt werden, wie komplex sich die Frage nach der richtigen Implementierung des scheinbar strukturell so einfachen Musters stellt. In Gesprächen mit Entwicklern, die sich erst in die Musterproblematik einarbeiten oder erst einen oberflächlichen Kontakt mit dieser hatten, stellte ich oft fest, dass das Singletonmuster vor allen anderen als bekannt angegeben wurde.

Dabei ist es ein Muster, das große Probleme für das Design und für die Implementierung – wie hier gezeigt wurde – nach sich zieht. Was in diesem Artikel ausgelassen wurde, sind insbesondere die Fragen der genauen Lebenszeit des Singletonobjektes. Der Autor des Buches „Modern C++ Design", Andrei Alexandrescu, setzt sich mit dieser Problematik genauer auseinander, denn es ist eine weitere notwendige Dimension bei den Überlegungen zum Einsatz von Singletons. Die Verwendung dieses Musters sollte also gut überlegt sein. In den meisten Fällen, in denen ich Singletonimplementierungen in Projekten antraf, warfen diese Probleme auf und wären durch andere Konstrukte problemlos ersetzbar gewesen. Die wenigen Zeilen des Musterbeispiels und die vermeintlich einfache Struktur haben einen verführerischen Charakter. Das Singleton ist eine Lösung für ein exklusives, selten auftretendes Problem und sollte daher ebenso exklusiv und selten verwendet werden.

3.3 Datenstrukturen und Containerklassen

Die wesentlichen Domänen des objektorientierten Entwurfs sind einerseits die Identifikation von Klassen und andererseits die Modellierung der Beziehungen zwischen den Klassen. Gerade die Beziehungen zwischen den Klassen ermöglichen die flexiblen und erweiterbaren Datenstrukturen, die zur Entwicklung komplexer Software nötig sind. Mit welchen technischen Hilfsmitteln diese Beziehungen im Einzelnen modelliert werden, soll hier ausschnittsweise erläutert werden.

Insbesondere die 1-zu-n-Beziehungen sind es, die häufig vorkommen und im Sprachstandard selbst nur eine statische Repräsentation finden. Der folgende Code definiert zum Beispiel eine 1-zu-3-Komposition zwischen A und B:

Listing 3.10. Eine 1-zu-3-Komposition

```
class A
{
  // ...
};

class B
{
public:
  // ...
private:
  A a1, a2, a3;
};
```

Eine 1-zu-5-Aggregation lässt sich so realisieren:

Listing 3.11. Eine 1-zu-5-Aggregation

```
class A { /* ... */ };

class B
{
public:
  // ...
private:
  A* array[5];
};
```

Wenn n jedoch variabel sein soll, d. h. wenn zur Laufzeit sich das n der 1-zu-n-Relation ändern und an die aktuellen Notwendigkeiten anpassen soll, dann muss an der Stelle der direkten Komposition oder des Zeigerarrays eine Containerfunktionalität integriert werden.

Klassen, die solche 1-zu-n-Relationen ermöglichen, nennt man Containerklassen. Die Standardbibliothek enthält solche Containerklassen in der eingebetteten STL-Bibliothek[9]. An dieser Stelle jedoch soll das grundlegende Verhalten einer Containerklasse erarbeitet werden, ohne schon auf vorgefertigte Implementierungen zurückzugreifen. Es soll eine eigene Implementierung erarbeitet werden, um die dafür wichtigen Techniken kennenzulernen.

Dabei soll zuerst auf die „natürlichen Grundtypen" von Datencontainern, die Liste, den Vektor und den Binären Baum, eingegangen werden. Jeder dieser drei Grundtypen hat bestimmte Vorteile gegenüber den anderen. Man wählt den, der aufgrund seiner internen Datenhaltung dem zu lösenden Problem am meisten entgegenkommt.

3.3.1 Die Liste

Die Liste ist ganz anders organisiert als ein Vektor oder ein Array. Sie verkettet ihre Elemente und allokiert damit immer genau so viel Speicher, wie es für die Anzahl der Elemente in ihr und die dazugehörigen Verwaltungsdaten notwendig ist. Die Verkettung der Elemente wird dadurch erreicht, dass für jedes Element in der Liste ein Knotenobjekt existiert, das das Element aufnimmt und einen Zeiger auf den nachfolgenden Knoten enthält – im Fall der doppelt verketteten Liste enthält der Knoten auch einen Zeiger auf den vorangehenden Knoten. Wenn ein Element in die Liste eingefügt wird, wird dafür ein neues Knotenobjekt angelegt, das Element damit „verpackt" und der neue Knoten in die bestehende Kette aufgenommen. Dazu sind nur einige

[9] Siehe Abschnitt 5.2.6 beginnend auf Seite 292.

wenige, in ihrer Anzahl konstante Zeigeroperationen notwendig. Das Einfügen eines Elementes in die Liste braucht immer die gleiche Zeit, ganz egal, wie viele Elemente die Liste schon enthält. Der Benutzer der Liste bekommt im Idealfall von dem Knotenobjekt nichts mit, da dieses nur für das interne Funktionieren der Liste von Relevanz ist. Wichtig für den Benutzer ist allerdings die Schnittstelle, mit der die Elemente der Liste ausgelesen werden können. Dort gibt es grundsätzlich zwei Möglichkeiten: die erste besteht darin, der Liste selbst Zugriffsmethoden zu geben, die ein sequenzielles Durchlaufen der Listenelemente ermöglichen. Dann muss allerdings ein interner Zeiger mitgeführt werden, der den aktuellen Status der Traversierung speichert; d. h. dieser Zeiger zeigt auf das Element, das der aktuellen Position der Traversierung entspricht. Möchte man nun wahlweise eine neue sequenzielle Traversierung beginnen und die alte unterbrechen und später fortsetzen, lässt sich das mit der angedeuteten Variante nicht lösen. Es kann mit dieser Implementierung immer nur eine Traversierung durchgeführt werden; wenn eine andere begonnen werden soll, muss die erste vollständig abgebrochen werden. Die zweite Variante, den Zugriff auf die Listenelemente zu modellieren, besteht darin, den Zugriff als solchen zu modellieren. Der „Zugriff" ist ein Substantiv – was durch den Artikel ausgedrückt wird – und lässt sich deshalb als Klasse darstellen. Modelliert man für den Zugriff einen eigenen Typ, so lässt sich davon – wie in der Sprache auch – die Mehrzahl bilden. Es können dann mehrere Zugriffe gleichzeitig instanziiert werden (stattfinden). Eine solche Implementierung soll hier durchgesprochen werden.

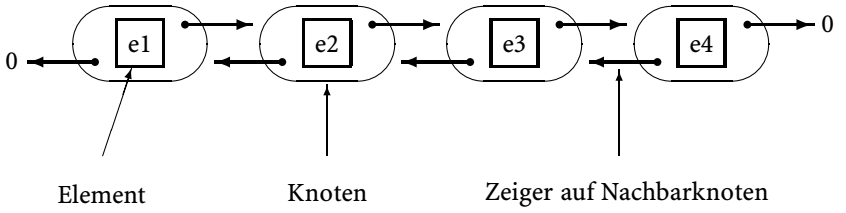


Abbildung 3.4. Die Organisation einer doppelt verketteten Liste

Gehen wir also in einem ersten Schritt für die Implementierung von einem einfachen Fall aus und schreiben eine Liste, die Elemente vom Typ `int` aufnehmen soll. Ohne den Zugriff zu betrachten, braucht die Liste drei Datentypen: das Element, den Knoten und die Liste selbst.

Listing 3.12. Liste Knoten und Element

```
typedef int ELEMENT;

class Knoten
{
};

class Liste
{
};
```

Der Einfachheit halber implementieren wir gleich eine doppelt verkettete Liste[10]. Der Knoten muss also zwei Zeiger und das Element enthalten. Natürlich braucht er auch einen Konstruktor, um seine Attribute zu initialisieren.

Listing 3.13. Die Implementierung des Knotens

```
class Knoten
{
public:
  Knoten( ELEMENT e ) : element(e), prev(0), next(0) {}
  ELEMENT element;
  Knoten *prev, *next;
};
```

Da der Knoten nur von der Liste verwendet wird und in dieser auch zugegriffen werden soll, wird entweder die Sichtbarkeit der Attribute als öffentlich definiert, oder die Liste wird als `friend`-Klasse deklariert. Der Einfachheit halber wird die Knotenklasse zunächst in der Form des vorangegangenen Listings belassen. Die Klasse `Liste` muss nun einen Anfangs- und einen Endknoten enthalten, und diese auch initialisieren.

Listing 3.14. Die Attribute der Liste

```
class Liste
{
public:
  Liste() : anfang(0), ende(0) {}
private:
  Knoten *anfang, *ende;
};
```

Um das Problem mit dem Copy-Konstruktor zu eliminieren, sollte gleich ein solcher für beide Klassen deklariert und versteckt werden.

[10] Die einfach verkettete Liste stellt meistens höhere Anforderungen an die Implementierung als die doppelt verkettete.

Listing 3.15. Gesicherte Copy-Konstruktoren

```
class Knoten
{
  // ...
private:
  Knoten( const Knoten & ); // verstecken!
};

class Liste
{
  // ...
private:
  Liste( const Liste & ); // verstecken!
};
```

Wenn wir später zu der Überzeugung gelangen, dass ein gültiger Copy-Konstruktor für die Liste sinnvoll wäre, dann werden wir einen gültigen definieren. Das Verstecken des Copy-Konstruktors zu Beginn einer Modellierung ist ein kleiner Kniff, der viel Ärger ersparen kann, wie es in Abschnitt 2.2.21 auf Seite 89 schon erläutert wurde. Zurück zur Implementierung der Liste: Es können nun einige der Methoden der Liste definiert werden, die für die interne Datenhaltung verantwortlich sind. Im folgenden Listing wird eine Methode zum Anhängen eines Elements an das Ende der Liste definiert und der Destruktor, der die Liste korrekt aus dem Speicher entfernt.

Listing 3.16. Methoden der Liste

```
class Liste
{
public:
  Liste() : anfang(0), ende(0) {}
  ~Liste();
  void AddEnd( ELEMENT e );
private:
  Knoten *anfang, *ende;
  Liste( const Liste & ); // verstecken!
};

// ...

Liste::~Liste()
{
  Knoten *t = anfang;
  while( t )
  {
```

```
    Knoten *n = t->next;
    delete t;
    t = n;
  }
}

void Liste::AddEnd( ELEMENT e )
{
  Knoten *k = new Knoten( e );
  if( ende )
  {
    ende->next = k;
    k->prev = ende;
    ende = k;
  }
  else
  {
    anfang = ende = k;
  }
}
```

Damit sind die algorithmisch wichtigsten Teile für die Datenhaltung schon definiert. Eine Einfügeoperation am Anfang der Liste wird hier nicht vorgeführt und der Phantasie des Lesers überlassen. Natürlich sind auch noch viele andere Operationen auf die Liste denkbar. Kommen wir aber nun zum Zugriff. Der Zugriff muss auf einen Knoten in der Liste verweisen und über dessen Zeiger auch zu einem Nachbarknoten navigieren können. Außerdem muss er den Zugriff auf das Element ermöglichen.

Listing 3.17. Die Zugriffsklasse

```
class Zugriff
{
public:
  Zugriff( Knoten *pk ) : k(pk) {}
  void Inc() { k = k->next; }
  void Dec() { k = k->prev; }
  bool IsValid() const { return k != 0; }
private:
  Knoten *k;
};
```

Die Klasse `Zugriff` wird mit einem Knoten initialisiert. Da aber nur die Liste bis jetzt exklusiven Zugriff auf die Knoten hat, ist sie es, die den Zugriff initialisieren muss. Dafür kann sie Methoden zur Verfügung stellen.

Listing 3.18. Die Zugriffsschnittstelle der Liste

```
class Liste
{
public:
  Liste() : anfang(0), ende(0) {}
  ~Liste();
  void AddEnd( ELEMENT e );

  Zugriff Begin() { return Zugriff(anfang); }
  Zugriff End()   { return Zugriff(ende);   }
private:
  Knoten *anfang, *ende;
  Liste( const Liste & ); // verstecken!
};
```

Die Klasse `Zugriff` wird also durch die Klasse `Liste` instanziiert und nach außen kopiert. Das heißt, dass die Klasse `Zugriff` auch einen Copy-Konstruktor braucht. Der vom Compiler erzeugte Copy-Konstruktor genügt in diesem Fall, da durch die Bytekopie der Knotenzeiger korrekt kopiert wird[11]. Der breiteren Verwendung wegen könnte aber noch ein Standardkonstruktor definiert werden, der einen unspezifizierten Zugriff initialisiert. Ein solcher unspezifizierter Zugriff könnte später durch eine Zuweisung auf eine Liste gerichtet werden. Wieder ist es der generierte Zugriffsoperator, der in diesem (Ausnahme-) Fall die richtige Operation durchführt – eine Zuweisung Byte für Byte.

Listing 3.19. Implementierung der Zugriffsklasse

```
class Zugriff
{
public:
  Zugriff() : k(0) {}
  Zugriff( Knoten *pk ) : k(pk) {}
  void Inc() { k = k->next; }
  void Dec() { k = k->prev; }
  bool IsValid() const { return k != 0; }
  ELEMENT get() const { return k->element; }
private:
  Knoten *k;
};
```

Jetzt sollte die Liste mit einer Basisfunktionalität einsetzbar sein:

[11] Das ist eine der wenigen Situationen, in denen ausgerechnet eine Klasse, die einen Zeiger enthält, mit dem generierten Copy-Konstruktor auskommt und diesen auch tatsächlich verwendet.

Listing 3.20. Anwendung der Liste

```
// ...
#include <iostream>

int main()
{
  Liste liste;

  liste.AddEnd( 3 );
  liste.AddEnd( 5 );
  liste.AddEnd( 7 );

  Zugriff z1 = liste.Begin();
  Zugriff z2 = liste.End();
  while( z1.IsValid() && z2.IsValid() )
  {
    std::cout << z1.get() << "\t"
              << z2.get() << std::endl;

    z1.Inc();
    z2.Dec();
  }

  return 0;
}
```

Die Ausgabe des Programms sollte folgendermaßen aussehen:

```
3   7
5   5
7   3
```

Bis jetzt wurde der Elementtyp mit `int` spezifiziert. Damit man einen Container – bzw. diese Liste – breiter einsetzen kann, könnte die Liste einen Zeiger auf ein allgemeines Interface enthalten. Wenn man dann auf die Inhalte der Liste zugreifen möchte, muss man allerdings konvertieren.

Listing 3.21. Eine Basisklasse für polymorphe Listenelemente

```
// In Liste.h
class Listenelement
{
public:
  virtual ~Listenelement();
};
```

```
// In Liste.cpp
Listenelement::~Listenelement() {}
```

Dafür müssen natürlich auch alle Funktionen und Typen umgeschrieben werden, die mit dem Typ `ELEMENT` im Zusammenhang standen. Also:

Listing 3.22. Die Implementierung der Methode `AddEnd()` für polymorphe Listenelemente

```
void Liste::AddEnd( Listenelement *e )
{
  Knoten *k = new Knoten( e );
  if( ende )
  {
    ende->next = k;
    k->prev = ende;
    ende = k;
  }
  else
  {
    anfang = ende = k;
  }
}
```

Damit für die Klassen Knoten und die Klasse Zugriff nicht zu viele Änderungen durchgeführt werden müssen, wird folgende Technik angewandt:

Listing 3.23. Verschachtelte Klassen in der Listenimplementierung

```
// Liste.h
#ifndef __LISTE_H
#define __LISTE_H
class Listenelement
{
public:
  virtual ~Listenelement();
};

class Liste
{
private:
  typedef Listenelement* ELEMENT;

  class Knoten
  {
  public:
```

```
    Knoten( ELEMENT e )
    : element(e), prev(0), next(0) {}
    ELEMENT element;
    Knoten *prev, *next;
  };

public:
  class Zugriff
  {
  public:
    Zugriff() : k(0) {}
    Zugriff( Knoten *pk ) : k(pk) {}
    void Inc() { k = k->next; }
    void Dec() { k = k->prev; }
    bool IsValid() const { return k != 0; }
    ELEMENT get() const { return k->element; }
  private:
    Knoten *k;
  };
  friend class Zugriff; // Für die Verwendung von Knoten

  Liste() : anfang(0), ende(0) {}
  ~Liste();
  void AddEnd( Listenelement *e );

  Zugriff Begin() { return Zugriff(anfang); }
  Zugriff End()   { return Zugriff(ende);   }
private:
  Knoten *anfang, *ende;
  Liste( const Liste & ); // verstecken!
};
#endif // __LISTE_H
// Ende von Liste.h
```

Listing 3.24. Die Implementierung der Liste

```
// Liste.cpp
#include "Liste.h"

Listenelement::~Listenelement() {}

Liste::~Liste()
{
  Knoten *t = anfang;
  while( t )
```

```
  {
    Knoten *n = t->next;
    delete t;
    t = n;
  }
}

void Liste::AddEnd( Listenelement *e )
{
  Knoten *k = new Knoten( e );
  if( ende )
  {
    ende->next = k;
    k->prev = ende;
    ende = k;
  }
  else
  {
    anfang = ende = k;
  }
}
// Ende von Liste.cpp
```

Ein weiterer großer Vorteil dieser Implementierung ist es, dass die Klasse `Knoten` in der Liste versteckt ist, in der sie gebraucht wird. Immerhin ist es ja denkbar, dass es noch Listenklassen anderer Elementtypen geben kann. Diese brauchen auch ihre Knoten, die dann in einem Namenskonflikt stehen würden. Mit dem Zugriff verhält es sich ähnlich. Er ist abhängig vom Knotentyp, der wiederum abhängig vom Elementtyp ist. Die Klasse `Zugriff` genau in den Container zu verschachteln, für den sie gemacht wurde, ordnet die Zugehörigkeiten in natürlicher Weise.

Die in diesem Abschnitt vorgestellte Implementierungslösung einer Liste basiert auf dem Prinzip des Polymorphismus. Es können nur Elemente in die Liste aufgenommen werden, die von der Schnittstellenklasse `Listenelement` erben. Beim Auslesen der Liste entsteht damit das Problem, dass ein Downcast durchgeführt werden muss. Es muss der gespeicherte Zeiger in Richtung der reellen Kindklasse gecastet werden. Das birgt einige Risiken[12], die man besser vermeiden sollte. Eine weitere Einschränkung der zuletzt gezeigten Implementierung ist, dass sie nur für Elemente von Klassentypen verwendbar ist und keine Standardtypen aufnehmen kann, da diese nicht von `Listenelement` erben können. Im Abschnitt 4.1.14 ab Seite 251 wird die Implementierung der Liste mit Hilfe von Templates so abgeändert, dass die beschriebenen Einschränkungen nicht mehr gelten.

[12] Siehe Abschnitt 2.2.29 auf Seite 114 ff.

3.3.2 Der Vektor

Der Vektor stellt einen zusammenhängenden Speicherbereich dar, der bei Bedarf vergrößert wird. Er kann deshalb sehr gut mit einem Index zugegriffen werden. Einfügeoperationen sind dafür aufwändiger. Das Anhängen neuer Elemente am Ende des Vektors ist solange billig, bis eine Vergrößerung seiner Kapazität durchgeführt werden muss. Ein Vektor hat eine Größe und eine Kapazität. Die Größe ist die Anzahl der Elemente, die sich zu einem bestimmten Zeitpunkt in ihm befinden. Die Kapazität beschreibt den zu einem Zeitpunkt allokierten Speicher in Elementgrößen, d. h. die Kapazität ist diejenige Anzahl von Elementen, die ohne eine Neuallokation von Speicher in den Vektor passen würde. Wenn die Kapazitätsgrenze erreicht ist und ein neues Element in den Vektor eingefügt werden soll, dann muss ein neuer Speicherbereich allokiert werden, der um ein bestimmtes Delta größer ist als die alte Kapazität. Danach muss der gesamte alte Speicherbereich in den neuen umkopiert und freigegeben werden. Einfügeoperationen in einen Vektor an der Kapazitätsgrenze sind also teuer.

Listing 3.25. Eine Vektor-Implementierung

```
typedef int ELEMENT;

class Vektor
{
public:
  Vektor()
  : size(0),
    capacity(STANDARDCAPACITY),
    delta(STANDARDDELTA)
    { Init();}
  Vektor( unsigned k = 10, unsigned d = 10 )
  : size(0),
    capacity(k>0?k:1),
    delta(d>0?d:1)
    { Init();}

  ~Vektor() { delete [] buf; }

  void AddEnd( ELEMENT e );

  ELEMENT GetAt( unsigned idx ) const
  {
    return buf[idx];
  }
```

```
  ELEMENT operator[]( unsigned idx ) const
  {
    return buf[idx];
  }

  unsigned Size() const { return size; }

private:
  unsigned size;
  unsigned capacity;
  unsigned delta;
  ELEMENT *buf;
  enum { STANDARDCAPACITY = 10 };
  enum { STANDARDDELTA = 10      };
  void Init();
  void Grow();
};

void Vektor::Init()
{
  buf = new ELEMENT[ capacity ];
}

void Vektor::AddEnd( ELEMENT e )
{
  if( size == capacity )
    Grow();
  buf[size++] = e;
}

void Vektor::Grow()
{
  unsigned newcapacity = capacity + delta;
  ELEMENT *newbuf = new ELEMENT[newcapacity];
  for( unsigned i = 0; i < size; ++i )
    newbuf[i] = buf[i];
  delete [] buf;
  buf = newbuf;
  capacity = newcapacity;
}
```

Die Wahl des für ein Problem richtigen Containers hängt einerseits von den verwendeten Algorithmen und dem Speicherverbrauch ab, andererseits auch von seinem Verhalten der Speicherverwaltung des Betriebssystems ge-

genüber. Bei einer großen Anzahl von Systemen, die auch heute noch mit physikalischer Speicheradressierung arbeiten, kann eine Liste oder ein schlecht eingestellter Vektor zu einer unakzeptablen Fragmentierung des freien Speichers führen. Dieses Thema ist zwar kein direkt immanentes C++-Thema, stellt aber einen wichtigen Aspekt des Umfelds dar, in dem die C++-Entwicklung stattfindet. Neben Speicherverbrauch muss die Fragmentierung als Auswahlkriterium für den richtigen Container herangezogen werden. Moderne Betriebssysteme, die eine virtuelle Speicherverwaltung besitzen, entschärfen das Problem, beseitigen es aber nicht immer vollständig.

3.3.3 Dynamische Container und das Problem der Speicherfragmentierung

Bei der Nutzung dynamischer Datencontainer nutzt man automatisch dynamische Allokation und Deallokation des Hauptspeichers. Wenn man sich in einem entsprechend kritischen Umfeld bewegt, sollte das Problem der Fragmentierung des Speichers im Zusammenhang mit den eingesetzten Containern durchdacht werden. Dazu einige Überlegungen, die die Wirkungsweise verschiedener Container auf die Fragmentierung erklären. Da die Fragmentierung vor allem bei Systemen mit physikalischer Speicherverwaltung negativ in Erscheinung tritt, soll für diesen Abschnitt eine solche angenommen werden.

In der STL haben wir bezüglich ihrer Allokationsstrategie zwei grundsätzlich verschiedene Containerarten: das sind einerseits die Container, die mit Hilfe von verlinkten Verwaltungsstrukturen – den Knoten – die Elemente einzeln aufnehmen (`std::list`,`std::set`,`std::map`...) und andererseits der `std::vector` (und auch der `std::string`, wenn man ihn zur STL zählen möchte), der Platz für mehrere potenzielle Elemente in Blöcken vorallokiert.

Da sich Listen, Sets und die Derivate sehr gut für Einfüge- und Löschoperationen eignen, werden diese in vielen Fällen auch ausgeführt werden. Einfügeoperationen führen dabei zu Allokationen von Blöcken der Größe der verwendeten Verwaltungsstrukturen – Knoten. Löschoperationen entfernen gerade diese zumeist kleinen Blöcke und hinterlassen ebenso kleine Lücken. Intensives Arbeiten mit solchen Containern kann dazu führen, dass der Speicher in sehr viele kleine Fragmente zerteilt ist. Allerdings ist die Speicherverwaltung auch in der Lage, diese Lücken bei neuen Einfügeoperationen wieder zu schließen[13].

[13] Vorausgesetzt, die Speicherverwaltung setzt eine „first fit"- oder „best fit"- Strategie ein. Dazu werden die freien Speicherblöcke durch die Speicherverwaltung verkettet. Eine Allokationsanfrage geht die Kette entlang und nimmt sich den ersten Block, der passt (first fit), bzw. den Block, der am besten passt (best fit). Für die best-fit-Strategie wird die Liste der freien Blöcke meistens nach Größe sortiert.

Demgegenüber steht ein ganz anderes Fragmentierungsbild, das durch Vektoren verursacht werden kann. Wenn die Größe eines Vektors sehr gut eingestellt ist, muss er möglicherweise nicht zur Laufzeit reallokiert werden. Für die Verwendung eines Vektors aus der STL spricht aber gerade die Dynamik, da ja ansonsten ein einfaches Array mit fester Größe allokiert werden könnte. Man geht also vom Reallokationsfall aus. Dieser tritt ein, wenn mehr Elemente als ursprünglich zum Zeitpunkt der Einstellung der Größe geplant waren, in den Vektor aufgenommen werden sollen. In diesem Fall allokiert er einen Block mit seiner aktuellen Größe plus einem Delta und kopiert seinen gesamten Inhalt in den neuen Block um. Der alte Block wird danach deallokiert. Stellen wir uns vor, der Vektor wächst weiter und wird noch einmal vergrößert. Er allokiert wieder einen Block mit seiner neuen Größe plus einem Delta und führt die schon beschriebenen Schritte aus. Der neue Block ist so groß, dass er auf keinen Fall in den freien Bereich passt, den der Vektor bei der ersten Vergrößerung im Speicher hinterlassen hat. Bei der erneuten Vergrößerung hinterlässt er wieder einen relativ großen freien Block im Speicher. Bei intensiver Nutzung von Vektoren werden also viel weniger Fragmente des Speichers entstehen. Diese sind aber größer. Die Verwendung von Vektoren kann dazu führen, dass Speicher großräumiger zerteilt wird. Das kann sich dann besonders negativ auswirken, wenn wieder große Blöcke allokiert werden müssen.

Insbesondere bei der Entwicklung von Embedded-Systemen, die mit Speicherknappkeit oder mit einer physikalischen Speicherverwaltung zurechtkommen muss, gewinnen diese Überlegungen an Bedeutung. Dieses Problem muss ebenfalls durchdacht werden, wenn große Datenmengen mit der STL bearbeitet werden sollen. Um die Art der Speichernutzung bestimmter Container durch eigene Speicherverwaltungen zu verändern, können so genannte Allokatoren geschrieben werden. Diese sind aber nicht trivial. Wer sich damit auseinandersetzen möchte, sollte sich mit dem C++-Standard [ISO-C++] beschäftigen und eventuell auch damit, was der aktuelle Compiler daraus unterstützt.

3.3.4 Verfeinerung des Zugriffs durch überladene Operatoren

Den Zugriff auf einen Container, wie z. B. die Liste, kann man durch das Überladen von Operatoren noch verfeinern. Betrachten wir die Schnittstelle der Klasse `Zugriff` aus der Klasse `Liste`.

Listing 3.26. Einsatz von überladenen Operatoren für den Elementzugriff

```
class Zugriff
{
public:
  Zugriff() : k(0) {}
```

```
  Zugriff( Knoten *pk ) : k(pk) {}
  void Inc() { k = k->next; }
  void Dec() { k = k->prev; }
  bool IsValid() const { return k != 0; }
  ELEMENT get() const { return k->element; }
private:
  Knoten *k;
};
```

Man könnte die Gültigkeitsüberprüfung durch einen Operator ermöglichen.

Listing 3.27. Typenkonvertierungsoperator für Gültigkeitsprüfung

```
operator bool() { return k != 0; }
```

Damit könnte man ein Objekt des Typs `Zugriff` direkt in eine Bedingung einbinden. Für `Inc()`, `Dec()` und `get()` lassen sich die Operatoren `++`, `--` und `*` überladen. Damit erhält der Zugriff eine ähnliche Semantik wie die eines Zeigers.

Listing 3.28. Operatoren für Dereferenzierung, Inkrement und Dekrement in der Zugriffsklasse

```
ELEMENT operator *() { return k->element; }
Zugriff& operator++() { Inc(); return *this; }
Zugriff  operator++(int)
{
  Zugriff tmp( *this );
  Inc();
  return tmp;
}
Zugriff& operator--() { Dec(); return *this; }
Zugriff  operator--(int)
{
  Zugriff tmp( *this );
  Dec();
  return tmp;
}
```

In der C++-Standardbibliothek finden solche Operatoren Anwendung für die Iteratoren der STL[14].

[14] Siehe Abschnitt 5.2.6 auf Seite 292.

3.4 Arbeiten mit Invarianten

Eine für die Objektorientierte Programmierung wesentliche Methode zur Arbeit mit Invarianten nennt sich „Design by Contract“[15]. Die praktische Umsetzung dieser Methode ist in der Programmiersprache Eiffel realisiert. In C++ lässt sie sich größtenteils emulieren.

Die Methode Design by Contract ist ein formales Verfahren zur Beschreibung von Bedingungen für die Benutzung einer Klasse und zur Überprüfung von deren Korrektheit. Die Methode ist zunächst unabhängig von fachlichen Aspekten, bezieht diese aber für eine Beschreibung mit ein. Contract heißt Vertrag. In der Entwicklungsmethode Design by Contract werden formale Vereinbarungen formuliert zwischen Klassen und den die Klassen nutzenden Clientcodes. Der Vertrag regelt also die Beziehung zwischen Clientcode und Klasse. Doch was ist eigentlich ein Vertrag im Sinne von Design by Contract? Es ist einerseits die „Invariante“ einer Klasse und es sind andererseits die postulierten Vor- und Nachbedingungen der Methoden. Auch die Schnittstelle einer Klasse wird manchmal als Vertrag gesehen. Diese wird allerdings nicht zur Laufzeit, sondern zur Compilezeit überprüft.

Invarianten

Objekte enthalten Zustände. Je mehr Attribute eine Klasse hat, desto mehr rechnerische Zustände kann ein Objekt dieser Klasse annehmen. Dabei sind nicht alle Kombinationen von Attributwerten wünschenswert. Die Werte müssen im Sinne der Daten, die das Objekt repräsentiert, zueinander passend sein. Man spricht dann von einem konsistenten oder einem wohldefinierten Objektzustand. Die Methoden der Klasse müssen den Objektzustand konsistent bzw. wohldefiniert halten. Dafür lassen sich im speziellen Fall Regeln finden, wie sich die Attributwerte eines Objektes zueinander zu verhalten haben. Die Regel, die beschreibt, ob ein Objektzustand wohldefiniert und damit „gültig“ ist, bezeichnet man als Invariante. Eine solche Invariante muss abstrakt formulierbar sein. Der Konstruktor setzt diese Invariante in Kraft und der Destruktor macht sie ungültig. Mit der Invarianten ist auf keinen Fall das Regelwerk des fachlichen Codes gemeint, sondern eine allgemeine Regel, die möglichst einfach beschreibt, wann der Objektzustand gültig ist. Eine solche Invariante lässt sich an einem einfachen Beispiel sehr gut verdeutlichen (was in diesem Unterkapitel auch geschieht), ist aber für reale Objekte in realen Projekten manchmal sehr schwer zu finden. Insbesondere dann, wenn Klassen zu kompliziert sind, wenn sie zu viele Rollen spielen oder mit anderen Worten, wenn ihre Kohäsion zu gering ist, sind Invarianten nur sehr schwer zu formulieren. Auch eine zu starke Kopplung nach außen erschwert das Finden

[15] Der Begriff „Design by Contract“ ist ein registrierter Handelsname der Firma Eiffel Software.

einer Invarianten, denn mit der Kopplung nach außen erhöht sich auch der äußere Einfluss fremder Objekte auf den Objektzustand.

Das Formulieren von Invarianten öffnet auch einen formalisierten Zugang zur Überprüfung von Algorithmen. Wie eine Invariante für einen Objektzustand formuliert werden kann, so kann auch eine solche Invariante für einen Algorithmus gefunden werden. Wichtig für einen Algorithmus ist aber insbesondere das Formulieren von Vor- und Nachbedingungen, was man nicht mit der Invarianten selbst verwechseln sollte.

Für beide Fälle kann nun Testcode geschrieben werden, der die Bedingung der Invariante formal prüft, ohne auf algorithmische Besonderheiten des fachlichen Codes eingehen zu müssen. Wie das anhand von Klassen in C++ geschieht, wird im folgenden Abschnitt erklärt.

Überprüfung von Invarianten

Für das Finden einer Invariante sollte das Streben nach Einfachheit das oberste Gebot sein. Invarianten, die zu kompliziert sind, deuten darauf hin, dass das Klassendesign überarbeitet werden sollte. Wenn eine einfache Invariante für eine Klasse gefunden wurde, kann sich der algorithmische Code nach ihr richten. Das heißt, dass in jeder Methode vor und nach der Ausführung ihres Rumpfes die Invariante gelten muss. Während der Ausführung der Methode gilt die Invariante nicht! Der Grund dafür ist, dass die Methode, wenn sie nicht nur Zustände abfragt, selbstverständlich die Attributwerte des Objekts ändert. Innerhalb der Änderungssequenz treten nicht passende Attributwerte auf, die aber durch die Restsequenz und unter Zuhilfenahme lokaler Daten passend gemacht werden. Vor den Anweisungen einer Methode kann nun eine Überprüfung der Invariante gesetzt werden. Damit wird die Vorbedingung für die Ausführung der Methode geprüft. Nach der letzten fachlichen Anweisung kann die Invariante durch eine Nachbedingung überprüft werden.

Durch ein solches Vorgehen erhält man einen von speziellen fachlichen Erfordernissen unabhängigen Zugang zu möglichen Fehlern im Code. Der überprüfende Code verhält sich auch redundant zum fachlichen Code der Klasse, denn schließlich muss der fachliche Code die Einhaltung der Invarianten garantieren. Der Code der Vor- und Nachbedingungen überprüft nur, ob der fachliche Code sein Versprechen auch einhält, und reagiert je nachdem mit einem Abbruch des Programms, mit einem Trace, mit einer Exception oder sonstwie auf den Fehler des Codes. Die Redundanz des die Invariante überprüfenden Codes aus einem anderen, formalen Gesichtspunkt heraus ist das Schlüsselprinzip der vorgeschlagenen Technik.

Da man in einem performanten Code, den man dem Kunden ausliefert keine Redundanzen haben möchte, kann man verschiedene Techniken nutzen, die Redundanz abzuschalten. Bevorzugte Techniken sind einerseits die bedingte Compilierung und das Ausnutzen der Optimierung durch den Com-

piler und andererseits das Abschalten des Testcodes zur Laufzeit über das Setzen von Flags.

Beispielsweise kann für den Vektor, der im Abschnitt 3.3 über Datenstrukturen und Container vorgestellt wurde, eine Invariante formuliert werden:

1. Die Kapazität darf nicht kleiner 1 sein.
2. Die Größe ist immer kleiner oder gleich der Kapazität des Vektors.
3. Der Zeiger auf den Puffer muss einen korrekten Adresswert haben.

Die öffentlichen Methoden des Vektors können mit Vor- und Nachbedingungen abgesichert werden. Da innerhalb des Methodenrumpfs die Invariante nicht gilt, darf keine andere öffentliche Methode dort aufgerufen werden, denn diese könnte ja wieder Prüfcode enthalten, der dann fehlschlagen muss. Um die Methoden der Schnittstelle zu unterstützen sind die privaten Methoden da, die keine Prüfung der Invariante vornehmen.

Listing 3.29. Implementierung einer Vektormethode mit Invariantenprüfung

```
void Vektor::AddEnd( ELEMENT e )
{
  if( capacity < 1 || size > capacity || buf == 0 )
    throw SomeException();

  if( size == capacity )
    Grow();
  buf[size++] = e;

  if( capacity < 1 || size > capacity || buf == 0 )
    throw SomeException();
}
```

Die hier vorgestellte Methode des direkten Abprüfens der Invariante in einer `if`-Anweisung[16] hat den Nachteil, dass die Redundanz im ausgelieferten Compilat erhalten bleibt. Um die Prüfungen entfernen zu können, kann bedingte Compilierung eingeführt werden.

Listing 3.30. Invariantenprüfung durch entfernbaren Code

```
void Vektor::AddEnd( ELEMENT e )
{
#ifdef __DEBUG__
  if( capacity < 1 || size > capacity || buf == 0 )
    throw SomeException();
#endif

  if( size == capacity )
```

[16] Sie kann auch Wächterbedingung genannt werden.

```
    Grow();
  buf[size++] = e;

#ifdef __DEBUG__
  if( capacity < 1 || size > capacity || buf == 0 )
    throw SomeException();
#endif
}
```

Wenn `__DEBUG__` nicht definiert ist, wird der Prüfcode durch den Compiler entfernt. Die Definition kann mit `#define __DEBUG__` oder in der Kommandozeile des Compileraufrufs (`-D __DEBUG__`) erfolgen. Eine solche Technik ist auch in der Standardbibliothek verfügbar. Man kann eine so genannte „Assertion" verwenden, um die Prüfung der Invariante vorzunehmen. Dazu gibt es das Makro `assert()` in der Headerdatei `<cassert>`. Das Makro `assert()` bekommt ein Argument. Falls dieses Argument `0` ist, bricht das Makro die Programmausführung ab. Dazu wird intern `abort()` aufgerufen. Zuvor wird noch der Dateiname und die Zeilennummer auf der Standardausgabe ausgegeben. Durch die Definition von `NDEBUG` wird das Makro deaktiviert und durch den Compiler aus dem Code entfernt.

Listing 3.31. Invariantenprüfung mit dem Makro `assert()`

```
void Vektor::AddEnd( ELEMENT e )
{
  assert(capacity < 1 || size > capacity || buf == 0);

  if( size == capacity )
    Grow();
  buf[size++] = e;

  assert(capacity < 1 || size > capacity || buf == 0);
}
```

Die Überprüfungen können auch in Klassenmethoden ausgelagert werden, um sie feiner einstellen zu können. So könnte der Vektor die private Methode `checkInvariant()` bekommen, die, wenn man sie inline definiert, für ein Auslieferungsrelease der Software herausoptimiert werden kann.

Listing 3.32. Eine separate Methode für den Invariantencheck

```
inline void Vektor::checkInvariant()
{
#ifdef __DEBUG__
  if( capacity < 1 || size > capacity || buf == 0 )
    throw SomeException();
#endif
}
```

Es gibt auch noch Templatetechniken, die man für das optimierbare Schreiben von redundantem Code verwenden kann. Der Abschnitt 3.4 auf Seite 218 im Unterkapitel über Templates (4.1) beschreibt diese Technik näher.

Vor- und Nachbedingung von Methoden

Neben den Invarianten können mit Assertions noch sogenannte Vor- und Nachbedingungen von Methoden überprüft werden. Da Methoden Änderungen am Objektzustand durchführen, sollten auch für sie Regeln definiert werden können, die vor und nach der Ausführung der Methoden gelten müssen. Die Regeln, die definieren, in welchem Zustand die Attribute des Objekts und die Parameter der Methode sein müssen, nennt man Vorbedingung. Nach dem Abschluss der Methode müssen die Attribute des Objekts in einem definierten Endzustand sein. Die Regeln, die diesen Endzustand beschreiben, nennt man Nachbedingung. Ähnlich wie bei den Invarianten kann es eventuell sehr schwierig oder aufwändig sein, Vor- und Nachbedingungen in einer Abfrage in C++ zu formulieren. Nicht jede Abfrage ergibt Sinn, wenn die Laufzeit- oder Speicherkosten der Überprüfung teurer sind als der zu überprüfende Codeabschnitt. In diesen Fällen kann es auch sinnvoll sein, Vor- und Nachbedingungen nur zum Teil zu überprüfen und sie ansonsten vollständig in Kommentaren zu vermerken. Da für eine Klasse eine Invariante definiert werden kann, die für jede öffentliche Methode gilt, und für jede einzelne Methode Vor- und Nachbedingungen, kann man die Invariante als die allgemeinere Regel betrachten. Wenn die Invariante sauber definiert ist, werden Methoden-spezifische Regeln oft überflüssig. Je länger und komplizierter Methoden allerdings werden, desto wichtiger werden auch deren Vor- und Nachbedingungen. Außerdem können in die Vorbedingungen auch die Daten des Clientcodes, also die Parameter der Methode, mit einbezogen werden.

Zur Überprüfung dieser Regeln stellt die Standardbibliothek das im vorangegangenen Abschnitt vorgestellte Makro `assert()` zur Verfügung. Es wird je nach Projektsituation zu entscheiden sein, ob die Funktionalität des `assert()`-Makros für das jeweilige Projekt ausreichend ist. Es lassen sich einige Techniken denken, die ähnlich anwendbar sind wie das `assert()`-Makro, aber zur Compile- und Laufzeit wesentlich anders reagieren. Ein paar Aspekte sollen hier angesprochen werden.

Selektives Aktivieren von Prüfcode

Da es sich bei dem überprüfenden Code um redundanten Code handelt, ist die Frage der Aktivierung und Deaktivierung von zentraler Bedeutung. Schließlich entscheidet man sich in einem Projekt unter anderem deshalb für C++, weil diese Sprache sehr performant ist, sich deterministisch verhält und außerdem ein Compilat erzeugt, das einen kleinen Footprint hat. Diese Vorteile würde man mit einem übergroßen Prüfcode vergeben. Der Prüfcode soll vor

allem Ergebnisse liefern, die das Entwicklungsteam zur Verbesserung des Codes und zum Finden eventueller Fehler benötigt. Der Anwender der Software wird in den seltensten Fällen mit Ausgaben des Prüfcodes etwas anzufangen wissen. Aus diesem Grund muss der überprüfende Code für das Kundenrelease abschaltbar sein.

Ein weiterer Aspekt ist, dass sich das Zeitverhalten von Code ändert, wenn Überprüfungen durchgeführt werden. Insbesondere wenn der Code mit mehreren Threads oder Prozessen[17] arbeitet, kann das Verhalten durch den Prüfcode drastisch verändert werden[18].

Wie in dem Abschnitt über die Invariante beschrieben, lässt sich der prüfende Code durch bedingte Compilierung abschalten. Das kann auch in einem Makro wie `assert()` geschehen. Wünschenswert ist aber auch häufig ein selektives An- und Abschalten von Prüfungen in verschiedenen Modulen zur Laufzeit. Dazu können Flags[19] definiert werden, die abgefragt werden, bevor eine Prüfung durchgeführt wird. Solche Flags lassen sich durch verschiedene Techniken an- oder abschalten. Häufig wird dazu ein paralleler Prozess aufgesetzt, der nur das Beobachten des Programmcodes zur Aufgabe hat. Insbesondere in Systemen, die keine Standardausgabe besitzen, ist ein solches Vorgehen notwendig. Dieser Prozess kommuniziert dann mit einem kleinen dazugelinkten Programmteil, der als Spion die Ausgaben des Programms zum Prozess schickt. Diesen Vorgang nennt man *tracen*. Man verwendet solche Traceausgaben dazu, etwas über den internen Zustand eines Programms zu erfahren, um daraus Schlüsse für eine Fehlerkorrektur ziehen zu können.

Ein Template für Assertions

Das Makro `assert()`, das die Standardbibiothek zur Verfügung stellt, ist noch nicht der Weisheit letzter Schluss. Bjarne Stroustrup schlägt in seinem Standardwerk über C++ [Stroustrup98] eine Technik vor, die Templates verwendet. Da diese Lösung inhaltlich sehr gut in das Kapitel passt, soll auf die Templatetechniken vorgegriffen werden, ohne jedoch jeden Aspekt dieser Technik hier zu erläutern. Im nachfolgenden Kapitel über die Templates wird auf alle verwendeten Techniken eingegangen. Zurück zu dem Vorschlag von Stroustrup. Dieses Template kann vom Compiler sehr gut optimiert werden. Es ist ein Funktionstemplate, wie es in Abschnitt 4.1.1 ab Seite 223 erklärt wird. Wenn zum Beispiel konstante Argumente übergeben werden, kann der Compiler

[17] Ein Prozess ist ein separater Ablaufstrang, also ein parallel laufender Code mit eigenem Speicherbereich. Ein Thread ist ebenfalls ein paralleler Ablaufstrang. Ein Thread hat allerdings keinen eigenen Speicherbereich. Siehe Abschnitt 6.3.2 auf Seite 382.

[18] In dem einen oder anderen Softwareprojekt kann es schon einmal vorkommen, dass Synchronisierungen paralleler Ablaufstränge nur durch die Verzögerung des Prüfcodes korrekt arbeiten. Schaltet man den Prüfcode ab, so werden erst die Fehler in der Synchronisierung sichtbar.

[19] Einfache Variablen, die einen boolschen „An“ und „Aus“-Wert kennen.

die Assertion vollständig entfernen. Außerdem kann die Ausnahme variiert werden, die im Fehlerfall geworfen wird.

Listing 3.33. Ein Template für Assertions

```
template <class E, class A>
inline void Assert( A assertion )
{
  if( !assertion ) throw E();
}
```

Dieses Template kann folgendermaßen angewendet werden:

Listing 3.34. Anwendung der Assertion-Templates

```
void f1( unsigned i)
{
  Assert<Error>(i != 0);

  // ...
}

void f2( unsigned i )
{
  Assert<Error>(NDEBUG || i != 0);

  // ...
}
```

In der Funktion `f1()` wird eine einfache Vorbedingung geprüft. Schlägt die Überprüfung fehl, wird eine Ausnahme vom Typ `Error` geworfen. In der Funktion `f2()` ist die Überprüfung der Vorbedingung durch `NDEBUG` abschaltbar. Wird `NDEBUG` mit 1 definiert, ist die Bedingung wahr, der zweite Teil der Bedingung wird dann definitiv nicht mehr ausgewertet. Da die Definition von `NDEBUG` schon zur Compilezeit feststeht, wird der zweite Teil der Bedingung schon durch den Compiler entfernt. Damit kann der Compiler auch die ganze Assertanweisung entfernen, denn sie ist als Inlinefunktion definiert.

Es kann nun wichtig sein, dass die Ausnahme, die von der Assertanweisung geworfen wird, erst zur Laufzeit bestimmt wird. Dafür schlägt Stroustrup ein weiteres Template vor, das nach dem ersten als Inlinefunktion implementiert ist, das Ausnahmeobjekt aber in der Parameterliste übernimmt.

Listing 3.35. Ein verfeinertes Assertion-Template

```
template <class E, class A>
inline void Assert( A assertion, E exception )
{
  if( !assertion ) throw exception;
}
```

Dieses Template hat den Vorteil, dass auch Ausnahmen verwendet werden können, deren Konstruktor Daten entgegennimmt. So können beispielsweise Informationen über den aktuellen Kontext an das Ausnahmeobjekt übergeben werden. Der Nachteil dieser Lösung ist aber, dass das Template nicht mehr vollständig aus dem Code heraus optimiert werden kann.

4 Generische und generative Programmierung mit C++

Mit der Einführung von Templates in C++ hielt auch gleich ein ganz anderes Programmier-Paradigma Einzug in die Sprache. Templates stellen nichts zur Verfügung, das die Objektorientierte Programmierung in irgendeiner Weise unterstützen würde. Man muss in der Templateprogrammierung mit Vorgehensweisen arbeiten, die nichts mit der Objektorientierung zu tun haben. Doch welche Vorgehensweisen sind es dann, die man in der Templateprogrammierung einsetzen kann?

Eisenäcker und Czarnecki haben diese Frage in ihrem Buch „Generative Programming. Methods, Tools and Applications." [CzarneckiEisenecker00] etwas genauer untersucht und einen theoretischen Unterbau erarbeitet. Deshalb seien an dieser Stelle einige wichtige Ideen aus der genannten Publikation dargestellt:

Die Templateprogrammierung wird auch gerne als „generische" Programmierung oder als „generative" Programmierung bezeichnet. Dabei bezeichnen die Adjektive generisch und generativ zwei unterschiedliche Aspekte. Generisch bezeichnet die Anpassbarkeit eines Moduls, einer Komponente oder einfach eines Stücks Software an unterschiedliche Gegebenheiten und Anforderungen. Generativ dagegen bedeutet die Möglichkeit, etwas automatisch erzeugen zu lassen. Die Entwicklung von Vorlagen zur Erzeugung weiteren Codes und deren Nutzung bezeichnet man als generative Programmierung.

Beide Aspekte sind Domänen der Templateprogrammierung mehr als der Objektorientierung. Während die Objektorientierung eine Technologie der Strukturierung von Software ist, ist die generische und die generative Programmierung eine Technologie um Wiederverwendbarkeit von Code zu erreichen. Auch wenn Wiederverwendbarkeit als Schlagwort in fast allen Lehrbüchern zur Objektorientierung genannt wird, finden sich doch nur sehr wenige objektorientierte Techniken, die speziell die Wiederverwendbarkeit zum Ziel haben. Die Wiederverwendbarkeit bleibt in objektorientierten Projekten meist auf dem Stand, auf dem sie in Projekten mit strukturierter Programmierung auch schon war. Anpassbare – generische – Codeteile sind zur Wiederverwendung geschrieben. Erzeugende – generative – Codeschablonen sind auch in ihrer Rolle zur Wiederverwendung geschrieben.

Da Templates in C++ zur Compilezeit abgearbeitet werden, beziehen sich sowohl die generischen wie auch die generativen Fähigkeiten auf diese Zeitphase. Es ist in C++ grundsätzlich der Compiler, der die generischen und generativen Fähigkeiten besitzen muss, die mit den Templates genutzt werden. Wie man diese Fähigkeiten strukturiert nutzt, haben die Autoren von [CzarneckiEisenecker00] auch ein Stück weit erarbeitet. Andrei Alexandrescu

ist mit seiner Publikation „Modern C++ Design. Generic Programming and Design Patterns Applied." [Alexandrescu01] auf diesem Weg noch ein Stück weiter gegangen und hat einige Methoden vorgeschlagen, wie Templates nutzbringend in Framework- und Applikationsentwicklung einzusetzen sind. Eines seiner bekanntesten Konzepte ist das „Policy Based Design", das im Prinzip auf einem Template-basierten Strategiemuster[1] beruht und einen aspektorientierten Ansatz in C++ einführt[2].

4.1 Templates

In diesem Abschnitt wird die Syntax der Template-Programmierung behandelt, ohne auf deren Bedeutung für den Systementwurf einzugehen. Diese Aufgabe übernimmt der Abschnitt 4.2 ab Seite 268. Hier werden also die wesentlichen Regeln für die Codierung aufgelistet und deren Syntax durchgesprochen, damit an späterer Stelle darauf verwiesen werden kann. Die Templates waren noch nicht von Anfang an Bestandteil der Sprache C++. Sie wurden erst Anfang der 90er Jahre in die Beschreibung der Sprache aufgenommen. Die Unterstützung durch die damaligen Compiler war noch eine ganz andere Sache. Von Anfang an hatten man in der Template-Programmierung mit den unterschiedlichen Interpretationen durch die Compiler zu kämpfen – ein Problem, das auch heute noch aktuell ist. In Abschnitt 4.2 ist beschrieben, dass die Template-Programmierung die eigentliche Technik zum Erreichen der Wiederverwendbarkeit in C++ ist. Genau dieser Punkt wird dadurch erschwert, dass die Unterstützung durch verschiedene Compiler nicht in einheitlicher Weise gegeben ist. Portierbaren Code mit Templates zu schreiben ist also immer noch schwierig und verlangt die Reduktion der eingesetzten Techniken auf den kleinsten gemeinsamen Nenner der durch die Compiler unterstützten Features. Die Standardisierung der Template-Techniken kam 1998 mit der ANSI/ISO-Spezifikation 14882, wobei die Templates einen großen Teil im Standardisierungsdokument für sich in Anspruch nahmen. Trotz dieser Standardisierung von C++ verhält es sich immer noch ähnlich wie in den Anfängen der Template-Programmierung in C++. Die handelsüblichen Compiler unterstützen die Templates in sehr unterschiedlicher Weise. Es kann zwar davon ausgegangen werden, dass ein gewisser Kernbereich durch die meisten der heute üblichen Compiler unterstützt wird, bei der Verwendung weiterführender Template-Techniken allerdings müssen bis heute die Fähigkeiten des Compilers mit in Betracht gezogen werden. Gerade einige weit verbreitete Compiler haben große Mängel in der Unterstützung der Templates.

[1] Siehe [GoF95].
[2] Siehe Abschnitt 4.3 auf Seite 273.

Aber nun zu der Frage, was Templates eigentlich sind: Templates sind Konstrukte, die zur Compilezeit abgearbeitet werden. Es sind Codeschablonen, die vom Compiler dazu genutzt werden, Code zu erzeugen, indem ein definierter Anteil ersetzt wird. Der Anteil, der in der Schablone variabel ist, kann entweder eine Konstante sein oder ein Typ. In den meisten Fällen werden es Typen sein, die Templates parametrisieren. Im folgenden Abschnitt wird mit den Funktionstemplates eine Abgrenzung zu den Makros durchgeführt, die ebenfalls Compilezeitkonstrukte darstellen.

4.1.1 Funktionstemplates

Ein sehr häufig bemühtes Beispiel in diesem Zusammenhang ist die `min()`-Funktion: eine Funktion, die von zwei übergebenen Werten den kleineren zurückgibt. Eine solche Funktion könnte man folgendermaßen implementieren:

Listing 4.1. Eine traditionelle Implementierung der Funktion `min()`

```
int min( int a, int b )
{
  if( a < b ) return a;
  return b;
}
```

oder

Listing 4.2. Eine weitere traditionelle Implementierung von `min()`

```
int min( int a, int b )
{
  return ( a < b ) ? a : b;
}
```

Das Problem mit einer solchen Implementierung ist, dass bis jetzt nur ein Typ existiert, auf den man diese Funktion anwenden kann. Natürlich kann man die Möglichkeiten der Funktionsüberladung nutzen, um weitere `min()`-Funktionen mit anderen Parametertypen zu definieren. Also:

Listing 4.3. Überladene Versionen von `min()`

```
unsigned min( unsigned a, unsigned b )
{
  return ( a < b ) ? a : b;
}
double min( double a, double b )
{
  return ( a < b ) ? a : b;
}
```

Man hat mit dieser Lösung allerdings nur eine endliche Anzahl von Typen abgedeckt. Es sind nicht alle Typen, von denen sich generell ein Größenvergleich durchführen ließe. Eine Lösung, die schon in C existierte, ist die Makroimplementierung der `min()`-Funktion:

Listing 4.4. `min()` als Makroimplementierung

```
#define min( a, b ) (((a)<(b))?(a):(b))
```

Die Klammerung wird dafür gebraucht, keine zufälligen Interferenzen zwischen den Parametern und den umgebenden Operatoren zu erzeugen. Aber gerade solche Interferenzen zwischen Parameter und Makro sind es, die das Makro zu einer tendenziell schlechten Implementierung machen. Übergibt man dem Makro `min()` die Parameter 4 und 5, wird man das korrekte Ergebnis 4 bekommen. Das folgende Beispiel liefert ein falsches Ergebnis:

Listing 4.5. Unerwünschter Nebeneffekt bei der Verwendung eines Makros

```
#include <iostream>

#define min( a, b ) (((a)<(b))?(a):(b))

int main()
{
  int a = 5;
  int b = 4;

  std::cout << min( a, ++b ) << std::endl;

  return 0;
}
```

Das Ergebnis einer Programmausführung ist 6. Das Problem kommt daher, dass Makros nicht typisieren und eine reine Textersetzung durchführen. Der Präprozessor macht also aus

`min(a,++b)`

die Sequenz

`(((a)<(++b))?(a):(++b))`.

Und darin taucht der Inkrementoperator zweimal auf.

Eine elegantere Lösung stellen die Funktionstempates dar. Für die `min()`-Funktion lässt sich ein Funktionstemplate schreiben, das den Compiler in die Lage versetzt, überladene Varianten der `min()`-Funktion zu generieren:

Listing 4.6. Template-Implementierung der Funktion `min()`

```
template <typename T>
T min( T a, T b )
{
```

```
  return (a < b) ? a : b;
}
```

Templates werden immer mit dem Schlüsselwort `template` eingeleitet. Das Schlüsselwort `typename` definiert einen Platzhalter für einen beliebigen Typ. Anstelle von `typename` kann man auch das Schlüsselwort `class` verwenden. Es hat an dieser Stelle exakt die gleiche Bedeutung und deklariert einfach einen Platzhalter für einen beliebigen Typ (nicht nur einen Klassentyp)[3].

Listing 4.7. Die alte Template-Syntax mit dem Schlüsselwort `class`

```
template <class T>
T min( T a, T b )
{
  return (a < b) ? a : b;
}
```

Man kann nun alle denkbaren Modifikationen an der Funktion anbringen. So kann man beispielsweise eine Referenzparameterübergabe definieren. Schließlich kann diese `min()`-Funktion auch mit Klassentypen angewendet werden, die den operator < definieren.

Listing 4.8. Referenzübergabe und Rückgabe bei Templatefunktion

```
template <class T>
T& min( T &a, T &b )
{
  return (a < b) ? a : b;
}
```

Funktionstemplates können auch den `inline`-Modifizierer enthalten und Vorlagen für Inlinecode bilden.

Listing 4.9. Template-Implementierung einer Inline-Funktion

```
template <class T>
inline const T& min( const T &a, const T &b )
{
  return (a < b) ? a : b;
}
```

In älteren C++-Standards konnten Funktionstemplates nicht überladen werden. Der Abschnitt über den Barton-Nackman-Trick 4.1.11 auf Seite 242 befasst sich mit dieser Problematik. Der aktuelle ANSI/ISO C++-Standard erlaubt das Überladen von Funktionstemplates und aktuelle Compiler haben im Allgemeinen auch keine Schwierigkeiten damit.

[3] Die Verwendung des Schlüsselwortes `class` an dieser Stelle ist älter als die Verwendung von `typename`, die erst mit dem ANSI/ISO-Standard von 1998 eingeführt wurde.

4.1.2 Klassentemplates

Die wichtigste und häufigste Anwendung der Templates in C++ sind die Klassentemplates. Wie die Methoden von den Typen ihrer Parameterlisten abhängig sind, so sind Klassen von ihren Attributtypen und Methodenparametern abhängig. Diese Typen können in Klassentemplates parametrisiert werden. Außerdem können in einem Klassentemplate auch konstante Werte als Templateparameter übergeben werden. Diese können nur mit den Standarddatentypen mit Ausnahme von `double` typisiert werden.

Listing 4.10. Ein Klassentemplate

```
template <typename T, unsigned n>
class Array
{
public:
    enum { size = n };
    T& operator[]( int i ) { return data[i]; }
private:
    T data[n];
};
```

Das Beispiel zeigt eine allgemeine Form eines Arrays, das dem Benutzer eine Bereichskontrolle zur Verfügung stellt. Das Template kann mit einem beliebigen Typ `T` instanziiert werden, der kopierbar ist und einen Standardkonstruktor besitzt. Damit stellt diese Arrayimplementierung keine höheren Anforderungen an den Inhaltstyp, als es herkömmliche C-Arrays tun. Im Übrigen bringt die Klasse auch keine Kosten bezüglich Speicherverbrauch mit, denn der Klammeroperator ist `inline`-definiert, `size` ist eine Konstante ohne Speicherplatz und es wurden keine virtuellen Methoden definiert.

Bei der Instanziierung des Templates müssen die Templateparameter in spitzen Klammern angegeben werden. Damit wird dem Compiler aufgetragen, einen reellen Typ aus der Schablone zu erzeugen.

Listing 4.11. Instanziierung eines Klassentemplates

```
// ...
Array<int, 5> a1;

for( int i = 0; i < a1.size; ++i )
  a1[i] = 0;
// ...
```

Das Array kann automatisch oder dynamisch angelegt werden:

Listing 4.12. Dynamische Instanziierung eines Objekts einer Templateklasse

```
// ...
Array<int, 5> *pa = new Array<int, 5>();

for( int i = 0; i < pa->size; ++i )
  (*pa)[i] = 0;
// ...
delete pa;
// ...
```

Klassen haben normalerweise nicht nur Inlinemethoden, sondern auch solche, die außerhalb der Klasse definiert werden. Für die Methoden von Templateklassen gibt es eine extra Templatedefinition, die auch unabhängig vom Klassentemplate definiert werden muss.

4.1.3 Methodentemplates

Das Beispiel des Arrays kann um einige Methoden erweitert werden. So kann das Array beispielsweise eine Methode `max()` enthalten, die den größten Wert aus der Sequenz zurückliefert. Ebenso könnte eine Methode `min()` definiert werden, die den kleinsten Wert des Arrays liefert.

Listing 4.13. Methoden in einem Klassentemplate

```
template <typename T, unsigned n>
class Array
{
public:
    enum { size = n };
    T& operator[]( int i ) { return data[i]; }
    const T& max() const;
    const T& min() const;
private:
    T data[n];
};
```

Im Klassentemplate befinden sich die Prototypen der Methoden, also nur deren Deklaration. Die Definition der Methoden erfolgt in separaten Templates:

Listing 4.14. Separate Templates für die Methodenimplementierungen

```
template <typename T, unsigned n>
const T& Array<T,n>::max() const
{
  const T* p = &data[0];
  for( int i = 1; i < size; ++i )
    if( *p < data[i] ) p = &data[i];
  return *p;
}

template <typename T, unsigned n>
const T& Array<T,n>::min() const
{
  const T* p = &data[0];
  for( int i = 1; i < size; ++i )
    if( *p > data[i] ) p = &data[i];
  return *p;
}
```

Indem an einem Objekt, dessen Typ ein instanziiertes Klassentemplate ist, eine Methode aufgerufen wird, sucht der Compiler nach einem passenden Methodentemplate und instanziiert dieses, wenn er es gefunden hat.

Listing 4.15. Anwendung der Template-Methoden

```
// ..
Array<int, 3> a1;

a1[0] = 4;
a1[1] = 9;
a1[2] = 2;

std::cout << a1.min() << std::endl;
std::cout << a1.max() << std::endl;
// ...
```

Der Compiler instanziiert das Methodentemplate wirklich erst, wenn im Quellcode der erste Aufruf der Methode gefunden wurde. Wird eine Methode nicht aufgerufen, dann wird auch das Template nicht instanziiert und verursacht auch keine Kosten im Compilat.

Ein Methodentemplate kann nicht spezialisiert werden.

4.1.4 Instanziierung von Templates

Templates werden immer bei der ersten Verwendung instanziiert. Wird ein Template nicht verwendet, dann generiert der Compiler auch nichts, was man in den Objektdateien finden könnte. Wie im vorangegangenen Abschnitt angedeutet, betrifft das auch die Methoden der Klassentemplates. Selbst wenn die Klasse instanziiert wird, werden die davon unabhängigen Methodentemplates nur dann instanziiert, wenn sie auch verwendet werden. Die Instanziierung der Klasse ohne die dazugehörigen Methoden nennt man auch die Instanziierung der Typeninformation. Die Typeninformation wird zum großen Teil nur während der Compilezeit gebraucht. Symbole in den Objektdateien werden nur dann angelegt, wenn dabei auch Methoden instanziiert werden oder die Klasse virtuelle Methoden besitzt. Im letzteren Fall müssen Symbole im Zusammenhang mit der V-Tabelle angelegt werden. Insgesamt garantiert dieses Verfahren der getrennten Instanziierung der Einzelelemente der Template-Klasse einen sparsamen Umgang mit Speicherressourcen und vermeidet unnötige Fehlermeldungen bei zu bestimmten Parametern unpassendem Code. Es hilft nicht zuletzt auch dabei, die Compilezeit zu verringern.

In bestimmten Fällen ist es aber nötig, den Compiler explizit zur Instanziierung eines bestimmten Templates zu veranlassen. Dazu gibt es die Syntax:

```
template class X<T>;
```

Wenn also die Templateinstanz `Array<int,5>` explizit instanziiert werden soll, so sieht das folgendermaßen aus:

Listing 4.16. Syntax der expliziten Templateinstanziierung

```
template class Array<int,5>;
```

Wann und warum solche expliziten Instanziierungen durchgeführt werden müssen, damit beschäftigt sich der Abschnitt 4.1.15 über die explizite Templateinstanziierung ab Seite 264. An dieser Stelle soll noch angemerkt werden, dass auch eine `typedef`-Anweisung zur Instanziierung eines Templates führen kann.

Listing 4.17. Instanziierung eines Templates durch eine Typendefinition

```
// ...
typedef Array<int,5> INTARRAY5;
// ...
```

Die Zeile führt dazu, dass der Compiler den Klassentyp instanziiert[4].

[4] In älteren C++-Standards wurden Typendefinitionen noch zur expliziten Instanziierung eingesetzt.

4.1.5 Member- Templates

Klassen können Templates als Elemente enthalten. Mit Templates können Methoden, verschachtelte Strukturen und innere Klassen implementiert werden. Für die beiden letztgenannten Templatearten gibt die umgebende Klasse einfach einen Namensraum ab. In Listing 4.18 ist eine Methode als Member-Template einer Templateklasse realisiert. Die Methode `copy_from()` kann als Parameter jeden beliebigen Containertypen aufnehmen, der wie die STL-Containertypen einen `const_iterator` hat und die Methoden `begin()` und `end()` besitzt. Natürlich muss auch der Inhaltstyp des übergebenen Containers zu dem des Arrays passen. Sie kopiert Elemente aus dem übergebenen Container in das Array, bis es voll ist. Wenn dazu die Elemente im Container nicht ausreichen, wird nur ein Teil des Arrays mit Containerwerten gefüllt.

Listing 4.18. Deklaration und Implementierung eines Membertemplates

```
template <typename T, unsigned n>
class Array
{
public:
  enum { size = n };

  // ...

  template <typename C >
       void copy_from( const C &c );
private:
    T data[n];
};

template<typename T, unsigned n>
 template<typename C>
 void Array<T,n>::copy_from( const C &c )
 {
   int i = 0;
   typename C::const_iterator it = c.begin();
   while( i < n && it != c.end() )
   {
     data[i++] = *it;
     ++it;
   }
 }
```

Interessant ist dabei das Template der Implementierung. Es werden einfach die äußere und die innere Templatespezifikation nacheinander aufgeführt. Natürlich kann ein Membertemplate auch an der Stelle implementiert werden, wo es deklariert ist. In der Standardbibliothek gibt es Stellen, an denen Member-Templates verwendet werden.

4.1.6 Spezialisierung von Templates

Das Klassentemplate `Array<typename T, unsigned n>`, wie es im Abschnitt 4.1.2 vorgestellt wurde, kann breit eingesetzt werden. Objekte des instanziierten Templates können automatisch oder dynamisch angelegt werden. Neben der Unabhängigkeit von einem lokalen Funktionsscope bringt die dynamische Allokation noch den Vorteil mit, dass der Aufrufstack nicht belastet wird. Insbesondere bei etwas größeren Arrays ist das ein wichtiger Punkt, den man nicht vergessen sollte. Vieles hängt natürlich auch vom eingesetzten Betriebssystem ab. Schreibt man aber Code, der zu sehr den Stack belastet, wird dieser Code auch schwerer zu portieren sein[5].

Es ist doch denkbar, dass in einem bestimmten Projekt sehr oft ein Array von einer spezifischen Größe gebraucht wird, das nicht mehr auf den Stack passt. Zur Verarbeitung der Bilddaten in einer Digitalkamera beispielsweise könnte ein Array gebraucht werden, das genau ein Megapixel Bilddaten enthalten kann. Es könnte nun wünschenswert sein, dass dieses Array sich selbst um die dynamische Allokation des Speichers kümmert. Dafür wird das Template für die speziellen Parameterausprägungen spezialisiert.

Listing 4.19. Vollständige Spezialisierung einer Templateklasse

```
#define MEGAPIXEL  (1024*1024)

template <>
class Array<unsigned long, MEGAPIXEL>
{
public:
  Array() { data = new unsigned long [size]; }
  ~Array() { delete [] data; }
  enum { size = MEGAPIXEL};
  unsigned long& operator[]( int i ) { return data[i]; }
  const unsigned long& max() const;
  const unsigned long& min() const;
```

[5] Vor allem im Embedded-Bereich muss man auch heute noch sehr auf eine schonende Verwendung des Stacks achten. Dort eingesetzte Betriebssysteme haben meistens eine fest eingestellte Stackgröße für einen Prozess, die sich zur Laufzeit nicht vergrößern lässt. Desktopsysteme wie Linux oder Windows NT/2000/XP haben eine dynamische Stackerweiterung zur Laufzeit.

```
private:
    unsigned long *data;
};

const unsigned long&
Array<unsigned long, MEGAPIXEL>::max() const
{
  const unsigned long* p = &data[0];
  for( int i = 1; i < size; ++i )
    if( *p < data[i] ) p = &data[i];
  return *p;
}

const unsigned long&
Array<unsigned long, MEGAPIXEL>::min() const
{
  const unsigned long* p = &data[0];
  for( int i = 1; i < size; ++i )
    if( *p > data[i] ) p = &data[i];
  return *p;
}
```

Es ist notwendig, alle Methoden des spezialisierten Templates einzeln zu definieren. Dabei wiederholt man nicht das Schlüsselwort `template <>`[6].

4.1.7 Partielle Spezialisierung

Betrachten wir noch einmal die Implementierung des Arrays, wie es im Abschnitt 4.1.2 vorgestellt wurde.

Listing 4.20. Allgemeines Template

```
template <typename T, unsigned n>
class Array
{
public:
    enum { size = n };
```

[6] Es ist eines der Geheimnisse des C++-Standards, warum an dieser Stelle der Definition von Methoden vollständig spezialisierter Klassentemplates dieses Schlüsselwort auch nicht erlaubt ist[7], obwohl es in der partiellen Spezialisierung (Abschnitt 4.1.7) sogar notwendig ist. Das ist eine der Inkonsistenzen, die das Erlernen der Templateprogrammierung in C++ so schwierig macht.

[7] Es gibt Compiler, die die Angabe von `template<>` erlauben, wie z. B. der C++ Compiler des Metrowerks Codewarrior. Diese Syntax ist allerdings nicht Teil des Standards.

```
    T& operator[]( int i ) { return data[i]; }
    const T& max() const;
    const T& min() const;
private:
    T data[n];
};
```

Die Methoden `max()` und `min()` ergeben für Datentypen Sinn, die vergleichbar sind. Manche sind das nicht. Beispielsweise ist es für Zeiger nicht wichtig, einen Größenvergleich durchzuführen. Man muss sie nur auf Identität prüfen können. Für Arrays aus Zeigern kann also auf diese Methoden verzichtet werden.

Dafür kann eine partielle Spezialisierung des Templates implementiert werden. Die partielle Spezialisierung unterscheidet sich von der vollständigen Spezialisierung dadurch, dass sie immer noch Parameter besitzt und in den spitzen Klammern an der Klasse die spezialisierte Form der Parameter aufführt.

Listing 4.21. Partielle Spezialisierung eines Templates

```
template <typename T, unsigned n>
class Array<T*,n>
{
public:
    enum { size = n };
    T& operator[]( int i ) { return data[i]; }
private:
    T data[n];
};
```

Bei der Anwendung des Templates wird nun durch den Compiler zwischen den beiden Implementierungen des Templates unterschieden.

Listing 4.22. Anwendung von Grundtemplate und spezialisiertem Template

```
// ...
Array<int, 5> a1;
Array<int*,5> a2;
// ...
```

An dem Array `a2` können nun keine Methoden `min()` und `max()` aufgerufen werden, denn die Schnittstelle ihres Arraytyps stellt sie nicht zur Verfügung.

Auf eine wichtige syntaktische Besonderheit muss an dieser Stelle noch eingegangen werden: auf die Templates von Methoden partiell spezialisierter Klassen. Die Methoden partiell spezialisierter Klassentemplates, die außerhalb der Klasse definiert sind, müssen als extra Templates definiert werden,

wie dies auch im allgemeinen Template der Fall ist. Das ist zunächst logisch, steht aber im Widerspruch zur vollständigen Spezialisierung, wo man für die Methoden zwar spitze Klammern mit den Templateparametern am Klassennamen anbringt, das Schlüsselwort `template` aber nicht auftaucht. Will man in der Projektarbeit mit Templates nicht verzweifeln, dann muss man diese Regel einfach wissen. Logisch herleiten kann man sie nicht. Also: *Die Definition von Methoden bei vollständiger Spezialisierung sieht anders aus als die bei partieller Spezialisierung.*

Hier die partielle Spezialisierung:

Listing 4.23. Methodentemplate mit partieller Spezialisierung

```
template <typename T1, typename T2>
class X
{
public:
  void f();
};

template <typename T1, typename T2>
void X<T1,T2>::f()
{
// Tu was...
}

template <typename T1>
class X<T1,int>
{
public:
  void f();
};

template <typename T1>
void X<T1,int>::f()
{
// Tu was...
}
```

Und zum Vergleich die vollständige Spezialisierung:

Listing 4.24. Vollständige Spezialisierung des Methodentemplates

```
template <>
class X<double,double>
{
public:
```

```
  void f();
};

void X<double,double>::f()
{
// Tu was...
}
```

Ich hebe diesen Punkt deshalb so hervor, weil er in der Programmierarbeit immer wieder zu Problemen führt und so manchem Programmierer das Leben schwer macht, auch wenn er an dieser Stelle zusammengefasst eher harmlos aussieht.

4.1.8 Vorgabeargumente für Templateparameter

In einer Template-Argumentenliste eines Klassentemplates können Vorgaben angegeben werden. Wie in der Parameterliste einer Funktion müssen die Vorgaben von hinten nach vorne erfolgen. Bei der Templateinstanziierung kann dann die Angabe der entsprechenden Argumente entfallen, wenn diese mit Vorgaben ausgestattet sind.

Listing 4.25. Vorgabeargument in der Templateparameterliste

```
template<typename T, template U = int>
class X { /*...*/ };

X<double> x1; // entspricht X<double,int>
X<double, char> x1;
```

So ist z. B. der String aus der C++-Standardbibliothek `std::string` folgendermaßen definiert:

```
typedef  basic_string<char> string;
```

Das zugrunde liegende Template `std::basic_string` hat aber drei Templateparameter und nicht nur eines. Dass es mit nur einem Argument instanziiert werden kann, liegt an den beiden Vorgabeargumenten.

```
template< class charT,
class traits = char_traits<charT>,
class Allocator = allocator<charT> >
class basic_string;
```

In Funktionstemplates können keine Vorgabeargumente angegeben werden.

4.1.9 Abhängige und qualifizierte Namen

In der Templateprogrammierung ist das Thema der Qualifizierung von Namen wichtig. Bevor jedoch auf den Problembereich eingegangen wird, einige grundsätzliche Bemerkungen zu Namen in C++. Ein Name, der alle Bezeichner enthält, damit er gefunden werden kann, und der keinen variablen Anteil eines noch unspezifizierten Templatearguments enthält, nennt man qualifiziert. Ein qualifizierter Name enthält demnach auch einen Zugriffsoperator (`.` oder `->`) oder den Scope-Operator (`::`).

Listing 4.26. Qualifizierung über den `this`-Zeiger

```
class Punkt
{
public:
  Punkt( int vx, int vy ) : x(vx), y(vy) {}
  bool operator==( const Punkt &p ) const
  {
    if( this->x == p.x && this->y == p.y )
      return true;
    return false;
  }
  int x;
  int y;
};
```

Die Namen `this->x` und `this->y` im Listing sind qualifiziert. Man hätte sie in diesem Beispiel auch ohne den `this`-Zeiger verwenden können. Sie wären durch den Compiler auch unqualifiziert gefunden worden. Beim Vergleich zweier Punkte kann der Vergleichsoperator angewendet werden.

Listing 4.27. Nichtqualifiziertes Einsetzen eines Operators

```
Punkt p1( 2, 3 );
Punkt p2( 3, 3 );
// ...
if( p1 == p2 )
{
  // ...
}
```

Der Vergleichsoperator wird hier unqualifiziert angegeben. Er ist demnach abhängig davon, ob der Compiler ihn nach seinen Lookupregeln findet. Man kann ihn auch qualifiziert angeben, wenn man möchte:

Listing 4.28. Qualifizierter Aufruf eines Operators

```
if( p1.operator==( p2 ) )
{
  // ...
}
```

Manchmal ist eine Qualifizierung auch zwingend nötig, um eine bestimmte Methode zuzugreifen. Betrachten wir einmal den folgenden allgemeinen Fall:

Listing 4.29. Überschriebene Methode

```
include <iostream>
class A
{
public:
  void f() { std::cout << "A::f()"´<< std::endl; }
};

class B : public A
{
public:
  void f() { std::cout << "B::f()"´<< std::endl; }
};

int main()
{
  B b;
  b.f();

  return 0;
}
```

Die Methode `f()` in `A` wird durch `f()` in `B` überschrieben. Allerdings bleibt `A::f()` grundsätzlich zugreifbar, denn schließlich haben wir keine Sichtbarkeitseinschränkung vorgenommen. Wenn also an einem Objekt der Klasse `B` die Methode aus `A` aufgerufen werden soll, muss qualifiziert werden:

Listing 4.30. Aufruf einer überschriebenen Basisklassenmethode durch Qualifizierung

```
...

int main()
{
  B b;
  b.A::f();

  return 0;
}
```

Aber nun zum Problem der Namen in Templates: In Templates kann der Compiler oft nicht das gleiche Lookupverfahren anwenden wie in Code, der nicht auf Templates basiert. Der Grund dafür ist, dass manche Namensräume ja erst nach der Instanziierung feststehen, die Namensauflösung aber während der Instanziierung erfolgen muss. Es gibt zwar Compiler, die durch eine bessere Voranalyse der Templates Namen auch in solchen Fällen noch größtenteils auflösen können, aber der Standard legt dort eine Regel fest, die gegen eine Auflösung in noch nicht existierenden Scopes spricht. Der folgende Code demonstriert das Problem:

Listing 4.31. Uneindeutiges Compileergebnis bei fehlender Qualifizierung in Templates

```
#include <iostream>

void f();

template<typename T>
class A
{
public:
  void f();
};

template<typename T>
class B : public A<T>
{
public:
  void g();
};

template<typename T>
void A<T>::f()
{
  std::cout << "A<T>::f()" << std::endl;
}

template<typename T>
void B<T>::g()
{
  f(); // Welches f() ist hier gemeint?
}

void f()
{
  std::cout << "Die globale Funktion f()" << std::endl;
```

```
}

int main()
{
  B<int> obj;

  obj.g();

  return 0;
}
```

An dieser Stelle muss nun etwas auf die Compilerhersteller eingegangen werden. Wenn man den oben stehenden Code mit dem Microsoft Visual C++ 7.1-Compiler übersetzt, erhält man die Ausgabe

```
A<T>::f()
```

Das gleiche geschieht, wenn man andere gängige Compiler, wie zum Beispiel den Borland C++ 5.5.1 oder den GNU C++ 3.3.1, heranzieht. Nimmt man den Metrowerks C++ 3.0, so stellt sich die Sache etwas komplizierter dar. Ohne Angaben spezieller Compilerschalter ergibt er das gleiche Ergebnis. Mit dem Compilerschalter `-iso_templates on` allerdings erhält man bei einem Programmlauf die Ausgabe

```
Die globale Funktion f()
```

Der ANSI/ISO Standard besagt, dass ein Lookup immer im Kontext eines Ausdrucks durchgeführt wird[8]. Da der Aufruf `f()` innerhalb eines Templates durchgeführt wird und eine Basisklasse `A<T>` noch nicht instanziiert ist, wird die globale Funktion `f()` durch den Lookup gefunden. Das entspricht auch dem Verhalten des Comeau C++-Compilers mit dem entsprechenden Compilerschalter `--A`, der standardkonformes Verhalten erzwingt. Wie andere C++-Compiler reagieren, kann im Einzelfall geprüft werden. Am GNU C++ wird in Richtung der besseren Unterstützung des C++-Standards viel gearbeitet, sodass auch bei diesem Compiler möglicherweise bald das Standardverhalten bei der Angabe des Schalters `--std=C++98` zu erwarten ist.

An dieser Stelle sieht man, dass der unbedachte Einsatz von nicht qualifizierten Namen in der Templateprogrammierung zu funktionalem Fehlverhalten oder nicht übersetzbarem Code führen kann. Der Aufruf von `f()` innerhalb von `g()` muss also qualifiziert werden.

[8] [ISO-C++] im Abschnitt 3.4.1 [basic.lookup.unqual]

Listing 4.32. Eindeutige Qualifizierung in Templates

```
#include <iostream>

void f();

template<typename T>
class A
{
public:
  void f();
};

template<typename T>
class B : public A<T>
{
public:
  void g();
};

template<typename T>
void A<T>::f()
{
  std::cout << "A<T>::f()" << std::endl;
}

template<typename T>
void B<T>::g()
{
  // Der Aufruf von f() aus der Elternklasse.
  this->f();
  A<T>::f(); // alternativ
  // Der Aufruf der globalen Funktion f():
  ::f();
}

void f()
{
  std::cout << "Die globale Funktion f()" << std::endl;
}

int main()
{
  B<int> obj;
```

```
  obj.g();

  return 0;
}
```

Bei diesem Code erzeugen alle genannten Compiler das gleiche Ergebnis, ganz egal, ob ein Schalter zur strikten Einhaltung des C++-Standards gesetzt ist oder nicht. Der Code ist standardkonform und portierbar. Aus dem gezeigten Grund ist es ratsam die Namen, die in Templates verwendet werden, soweit wie möglich zu qualifizieren.

4.1.10 Explizite Qualifizierung von Templates

Ein Klassen- oder Strukturtemplate kann neben anderen Elementen auch eingebettete Templates enthalten. Wenn ein solcher Bezeichner qualifiziert werden muss, kann man das mit einer der folgenden syntaktischen Formen tun:

```
::template
.template
->template
```

In der C++-Standardbibliothek wird eine solche Qualifizierung notwendig, wenn man mit dem Element `rebind` eines Allokators arbeitet. Ein Allokator definiert eine Speicherallokationsstrategie für einen bestimmten Typ `T`. Wenn man eine geänderte Allokationsstrategie benötigt, definiert man dafür einen weiteren Allokator. Innerhalb des Allokators gibt es ein Elementtemplate, das es ermöglicht, die gleiche Allokationsstrategie für einen beliebigen anderen Typ `U` zu bekommen.

Listing 4.33. Das `rebind`-Element aus der STL

```
template <typename T> allocator
{
  //...
  template <typename U> struct rebind
  {
    typedef allocator<U> other;
  };
  //...
};
```

Wenn man auf einen solchen Allokator zurückgreifen möchte, muss man den Bezeichner `rebind` als Template qualifizieren. Den verschachtelten Typ `other` qualifiziert man regulär mit dem Schlüsselwort `typename`.

Listing 4.34. Qualifizierung des `rebind`-Templates

```
template <class T, class A = std::allocator<T> >
class Container
{
  // ...
  class Node { /* ... */ };
  typedef
    typename A::template rebind<Node>::other
      NodeAllocator;
  // ...
};
```

Mit dem Element `rebind<>` erhält man also einen Allokatortyp mit der identischen Allokationsstrategie, aber für einen anderen Inhaltstyp. Datencontainer wie zum Beispiel verkettete Listen oder binäre Bäume können damit sowohl für die verwalteten Elemente, wie auch für den verwendeten Knotentyp die identische Allokationsstrategie zur Verfügung stellen.

4.1.11 Barton- Nackman- Trick

Gerade ältere Compiler haben Schwierigkeiten, Funktionstemplates zu überladen. In der Embedded-Softwareentwicklung kann es beispielsweise nötig sein, C++-Code auf einen Zielsystemcompiler anzupassen, der nach einem älteren C++-Standard arbeitet. AT&T-Standards definierten das Verbot Funktionstemplates zu überladen – etwas, was in ANSI/ISO-C++ erlaubt ist. Der folgende ANSI/ISO-Standard-konforme Code kann also mit älteren Compilern nicht übersetzt werden:

Listing 4.35. Der Problemcode

```
template <class T>
class X { /* ... */ };

template <class T>
class Y { /* ... */ };

template <class T>
void f( X<T> )
{
  // ...
}

template <class T>
void f( Y<T> )
```

```
{
  // ...
}
```

John J. Barton und Lee R. Nackman entwarfen 1994 eine Templatetechnik, die unter Ausnutzung eines Nebeneffektes das Problem löste. Diese Technik basiert darauf, dass `friend`-Funktionen bei ihrer `friend`-Deklaration innerhalb der Klasse auch definiert werden dürfen. Der folgende Code ist also gültig.

Listing 4.36. Implementierung einer `friend`-Funktion in einer Klasse

```
class Punkt
{
public:
  int x, y;
  Punkt( int vx, int vy ) : x(vx), y(vy) {}

  friend
  bool operator==( const Punkt &p1, const Punkt &p2 )
  {
    return (p1.x==p2.x) && (p1.y==p2.y);
  }
};
```

Die `friend`-Deklaration darf auch die erste Deklaration der Funktion überhaupt sein. Wenn sie dort auch definiert wird, wird es natürlich auch die einzige Deklaration bleiben. Was für eine Funktion gilt, ist übrigens für Klassen nicht möglich. Der folgende Code ist ungültig:

Listing 4.37. Syntaktischer Fehler

```
class X
{
  friend class Y {}; // ungültig!
};
```

Man kann also eine `friend`-Funktion samt Definition in eine Klasse aufnehmen. Es ist demnach auch möglich ein solches Konstrukt in ein Template aufzunehmen. Wird nun ein solches Klassentemplate instanziiert, erhält man quasi als Nebeneffekt die als `friend` definierte Funktion. Wenn die Funktion nun einen Parameter des Klassentyps besitzt, wird sie durch das Koenig-Lookup-Verfahren auch gefunden, denn die Klasse stellt den Namensraum der `friend`-Funktion dar.

Listing 4.38. Die Lösung

```
template <class T>
class X
{
  // ...
  friend void f( X<T> )
  {
    // ...
  }
};

template <class T>
class Y
{
  // ...
  friend void f( Y<T> )
  {
    // ...
  }
};
```

Für moderne Compiler ist dieser Trick nicht mehr notwendig. Er kann jedoch sehr sinnvoll sein, wenn man portablen Code schreiben möchte, der auch mit älteren Compilern übersetzbar bleiben soll.

4.1.12 Das Schlüsselwort `typename`

Mit dem Standard von 1998 hielt das Schlüsselwort `typename` Einzug in die Sprache C++. Zunächst kann es in einer Templatedefinition innerhalb der spitzen Klammern eingesetzt werden. Dort kann es das alte Schlüsselwort `class` ersetzen. Allerdings bleibt `class` genauso wie `typename` an dieser Stelle gültig. Es können also die beiden Alternativen verwendet werden:

entweder

Listing 4.39. Alte Syntax

```
template <class T>
class Etwas
{
  // ...
};
```

oder

Listing 4.40. Neue Syntax

```
template <typename T>
class Etwas
{
  // ...
};
```

Manche Entwickler verwenden das Schlüsselwort `typename` in den spitzen Templateklammern immer dann, wenn es sich um einen beliebigen Klassen- oder Standardtyp handeln kann. Das Schlüsselwort `class` verwenden sie, um zu zeigen, dass der Templateparameter im speziellen Fall nur ein Klassentyp sein darf. Der Compiler überprüft das natürlich nicht.

Die eigentliche Neuerung, die mit dem Schlüsselwort `typename` Einzug hielt, ist eine andere. Templates sind häufig von Typinformationen abhängig, die innerhalb der Templatedefinition nicht zur Verfügung stehen. Der folgende Code soll das verdeutlichen. Betrachten wir also eine Templatefunktion, die einen beliebigen Containerinhalt eines STL-Containers auf die Standardausgabe schreiben soll. Voraussetzung ist natürlich, dass für den Inhaltstyp ein Shiftoperator `<<` für `std::ostream` definiert ist. Andernfalls kann das Template nicht instanziiert werden und erzeugt einen Compilerfehler.

Listing 4.41. Nicht standardkonformer Template-Code

```
template <typename Container>
void print_out( const Container &c )
{
  // Die folgende Zeile ist nicht Standard konform!
  Container::const_iterator it = c.begin();
  while( it != c.end() )
  {
    std::cout << *it << std::endl;
    ++it;
  }
}
```

Ungeachtet der Tatsache, dass manche Compiler den Code übersetzen, ist er dennoch nicht standardkonform. Die Containerklassen der STL[9] haben einen internen Typ `const_iterator`. Der Code setzt diesen Typ einfach voraus. Würde dem Template ein Typ übergeben werden, der den internen Typ `const_iterator` nicht kennt, würde es in jedem Fall einen Compilerfehler geben. Das eigentliche Problem ist jedoch, dass die Notation

```
Container::const_iterator ...
```

[9] STL = Standard Template Library. Die STL ist Teil der C++-Standardbibliothek und enthält unter anderem Datencontainer. Siehe Abschnitt 5.2.6 auf Seite 292.

nur besagt, dass innerhalb der Klasse Container ein interner Bezeichner `const_iterator` existiert. Was dieser Bezeichner darstellt, ist beim Parsen des Templates nicht klar. Das Schlüsselwort `typename` wurde genau dafür in die Sprache aufgenommen, um anzuzeigen, dass es sich um einen Typnamen handelt. Der korrekte ISO-konforme Code muss also lauten:

Listing 4.42. Standardkonformes Anzeigen eines Typs

```
template <typename Container>
void print_out( const Cntainer &c )
{
  typename Container::const_iterator it = c.begin();
  while( it != c.end() )
  {
    std::cout << *it << std::endl;
    ++it;
  }
}
```

Man sollte sich in der praktischen Programmierung immer daran halten, Typnamen, die von Templateparametern abhängen, als solche zu bezeichnen. Vor allem dann, wenn man portablen Code schreiben möchte.

Um den Einsatz des Schlüsselwortes `typename` zu vereinfachen, sind hier vier Regeln aufgelistet:

1. Das Schlüsselwort `typename` kommt immer nur innerhalb von Templates vor. Außerhalb ergibt es keinen Sinn.
2. Der Name, an dem `typename` angebracht wird, ist abhängig von einem Template.
3. Es muss an einem qualifizierten Namen angebracht werden[10].
4. `typename` kommt nie in einer Liste der Spezifizierung von Basisklassen oder in einer Initialisierungsliste vor[11].

Um portablen Code zu schreiben, sollte auf das Schlüsselwort `typename` nicht verzichtet werden. Auch wenn die meisten heutigen Compiler Code in Templates ohne die expliziten Typdeklarationen akzeptieren, ist das nicht unbedingt auf eine verbesserte syntaktische Analyse zurückzuführen. Ein eventueller Fehler – eine Verwendung eines Bezeichners als Typ, obwohl er kein Typ ist – tritt dann bei der Instanziierung des entsprechenden Templates auf. Mit der Spezifikation als Typ kann der Compiler potentiell eine tiefere syntaktische Überprüfung von Templatecode durchführen, auch ohne dass Templates instanziiert werden müssten. Aus diesem Grund verlangt der ANSI/ISO-Standard die Verwendung des Schlüsselworts `typename` zur Kennzeichnung von Typen, die sich aus Templates ergeben.

[10] Siehe Abschnitt 4.1.9 Seite 236.

[11] Dort stehen per se nur Typenbezeichner. Eine Angabe des Schlüsselwortes an dieser Stelle wäre deshalb tautologisch.

4.1.13 Template-Templateparameter

Neben Typen können auch Klassentemplates als Parameter für Templates fungieren. Syntaktisch sieht das folgendermaßen aus:

Listing 4.43. Template-Templateparameter

```
template< template <typename> class T >
class X
{
};
```

Ein Template-Templateparameter muss ein Klassentemplate sein (es darf auch kein Template einer Struktur oder einer Union sein). Es müssen alle Templateparameter des zu übergebenden Templates aufgelistet werden. Im vorangegangenen Beispiel kann nur ein Template übergeben werden, das einen Typ entgegennimmt. Im folgenden Beispiel sind es zwei:

Listing 4.44. Template-Templateparameter

```
template< template <typename, typename> class T >
class X
{
};
```

Das Prinzip lässt sich auch verschachteln. Es können Parameter übergeben werden, die wiederum Template-Templateparameter erwarten.

Listing 4.45. Verschachtelte Template-Templateparameter

```
template< template <typename> class T >
class X
{
    T<int> t;
};

template< template
          <template <typename> class Q>
          class T >
class Y
{
};
```

Der Bezeichner `Q` wurde an dieser Stelle nur eingeführt, weil manche Compiler einen solchen Bezeichner erwarten, der hier allerdings nicht weiterverwendet werden kann. Standardkonform wäre das Beispiel auch ohne diesen Bezeich-

nernamen[12]. Solche Bezeichner können nur dazu verwendet werden, weitere Parameter innerhalb der gleichen Parameterliste zu definieren.

Listing 4.46. Vorgabetypen in Template-Templateparametern

```
template<typename T, typename S>
class A {};

template< template < typename Q,
                     typename R = Q*
                   > class T >
class X
{
    T<int> t;
};
```

Damit ist auch gezeigt, dass in der Parameterliste mit Vorgabewerten gearbeitet werden kann. Allerdings ist der Bezeichner `R` eigentlich wieder überflüssig, denn er wird ja nicht mehr weiter verwendet. Das Template `X` kann also auch ohne den Bezeichner `R` definiert werden.

Listing 4.47. Reduzierte Form

```
template< template < typename Q,
                     typename = Q*
                   > class T >
class X
{
    T<int> t;
};
```

Im folgenden Beispiel soll eine Templateklasse geschrieben werden, die ein Mapping zwischen Schlüsseln und Werten durchführt und eine lineare Suche zur Verfügung stellt. Neben Schlüssel- und Wertetyp soll ein Containertemplate übergeben werden, das intern zu zwei Containerklassen instanziiert wird. Der Parameter, der den Container festlegt, ist selbst ein Template mit einem Parameter. Das ist auch der Grund, warum keine STL-Container direkt übergeben werden können, denn diese besitzen zwei Templateparameter, z. B.:

```
template <class T, class Allocator = allocator<T> > vector;
template <class T, class Allocator = allocator<T> > list;
//...
```

[12] Überhaupt muss an dieser Stelle darauf hingewiesen werden, dass die beschriebene Spracheigenschaft der Template-Templateparameter von einigen Compilern nur sehr ungenügend unterstützt wird. Viele Compiler haben auch kleine Schwächen, die eine oder andere Ausdrucksform zu übersetzen, was das Schreiben portierbaren Codes mit dieser Technik momentan noch sehr erschwert.

Um das Problem zu umgehen, wurde in dem Beispiel ein kleiner Trick angewandt. Mit einem neuen Template und einer einfachen Vererbung kann ein neuer Containertyp erstellt werden, der nur einen Templateparameter entgegennimmt und deshalb an der gewünschten Stelle passt.

Listing 4.48. Beispiel eines generischen Mappings

```
#include <iostream>
#include <list>
#include <vector>
#include <string>

template <typename K, typename V,
          template <typename> class C>
class Mapping
{
public:
    void insert( const K& k, const V& v )
    {
        kc.push_back( k );
        vc.push_back( v );
    }
    const V* search( const K& k );
private:
    C<K> kc;
    C<V> vc;
};

template <typename K, typename V,
          template <typename> class C>
const V* Mapping<K,V,C>::search( const K& k )
{
    const V* p = 0;
    typename C<K>::const_iterator itk = kc.begin();
    typename C<V>::const_iterator itv = vc.begin();
    while( itk != kc.end() )
    {
        if( k == *itk )
        {
            p = &(*itv);
            break;
        }

        ++itk;
        ++itv;
```

```
    }
    return p;
}

// Trick zur Reduktion eines Templateparameters
template <typename T>
class MyList : public std::list<T> {};
template <typename T>
class MyVector : public std::vector<T> {};

int main()
{
    Mapping<int,std::string,MyVector> m;

    m.insert( 444, "Hans" );
    m.insert( 123, "Markus" );
    m.insert( 888, "Olaf" );
    m.insert( 100, "Robert" );
    std::cout << *(m.search( 888 )) << std::endl;

    return 0;
}
```

Es gibt auch eine andere Variante, die es erlaubt, STL-Container an dieser Stelle zu verwenden:

Listing 4.49. Anpassung von Template-Templateparametern für ein bestimmtes Übergabetemplate

```
template< typename K,
          typename V,
          template <
                    typename T,
                    typename = std::allocator<T>
                   > class C >
class Mapping;
```

Mit Hilfe eines Vorgabewertes kann der zweite Templateparameter für die STL-Containerklasse vordefiniert werden. Im Code muss deshalb nur ein Parameter für den Container angegeben werden. Das Beispiel kann also außer der Änderung in der Templatespezifikation so bleiben, wie es war. Bei der Instanziierung des Templates kann aber nun ein STL-Container übergeben werden:

Listing 4.50. Der Test des Mappings

```
int main()
{
    Mapping<int,std::string,std::list> m;
    //...
}
```

Die Lösung hat aber zwei Nachteile: Erstens wird durch den speziellen Vorgabetyp `std::allocator<T>` die Abhängigkeit zur STL erhöht und zweitens bewegen wir uns gerade auf dem Feld, auf das uns zu folgen so mancher Compiler noch gewisse Schwierigkeiten hat. Es bleibt nun abzuwarten, ob sich diese Schwierigkeiten in den nächsten Jahren legen werden.

4.1.14 Container mit Templates

Im Abschnitt 3.3 ab Seite 197 wurden die Grundprinzipien von Datencontainern besprochen. Dabei wurde unter anderem eine doppelt verkettete Liste entwickelt, die Objekte von verschiedenen Datentypen aufnehmen kann, die aber von der Schnittstellenklasse `Listenelement` erben müssen. Die Nachteile dieser Implementierung sind einerseits der Zwang, von der genannten Klasse abzuleiten, und andererseits die Unmöglichkeit, Standarddatentypen in der Liste aufzunehmen. Ein weiteres Problem ist auch, dass Objekte, die in die Liste aufgenommen werden sollen, nur in Form eines Zeigers abgespeichert werden. Das führt in den meisten Fällen zu dem Zwang, dass die entsprechenden Objekte dynamisch instanziiert sein müssen.

In diesem Abschnitt soll mit Hilfe der Templatetechnik eine Implementierung der Liste vorgestellt werden, die alle diese Nachteile nicht hat, die Objekte direkt abspeichern kann, keine Anforderungen an Elternklassen stellt, die aber auch Zeigerwerte und damit polymorphe Objekte aufnehmen kann.

Listing 4.51. Eine Templateimplementierung der verketteten Liste

```
// Liste.h
#ifndef __LISTE_H
#define __LISTE_H
template <typename T>
class Liste
{
private:
  typedef T ELEMENT;

  class Knoten
  {
  public:
```

```
    Knoten( ELEMENT e )
    : element(e), prev(0), next(0) {}
    ELEMENT element;
    Knoten *prev, *next;
  };

public:
  class Zugriff
  {
  public:
    Zugriff() : k(0) {}
    Zugriff( Knoten *pk ) : k(pk) {}
    void Inc() { k = k->next; }
    void Dec() { k = k->prev; }
    bool IsValid() const { return k != 0; }
    ELEMENT get() const { return k->element; }
  private:
    Knoten *k;
  };
  friend class Zugriff; // Für die Verwendung von Knoten

  Liste() : anfang(0), ende(0) {}
  ~Liste();
  void AddEnd( const T & );

  Zugriff Begin() { return Zugriff(anfang); }
  Zugriff End()   { return Zugriff(ende);   }
private:
  Knoten *anfang, *ende;
  Liste( const Liste & ); // verstecken!
};
template <typename T>
Liste<T>::~Liste()
{
  Knoten *t = anfang;
  while( t )
  {
    Knoten *n = t->next;
    delete t;
    t = n;
  }
}
```

```
template <typename T>
void Liste<T>::AddEnd( const T &e )
{
  Knoten *k = new Knoten( e );
  if( ende )
  {
    ende->next = k;
    k->prev = ende;
    ende = k;
  }
  else
  {
    anfang = ende = k;
  }
}
#endif // __LISTE_H
// Ende von Liste.h
```

Der ganze Sinn der Templatetechnik an dieser Stelle ist, dass keine Typenbindung mehr zur Laufzeit stattfindet, wie bei der vorhergehenden Implementierungsvariante, sondern zur Compilezeit.

Listing 4.52. Die Anwendung der Liste

```
#include <iostream>
#include "Liste.h"

int main()
{
  Liste<int> liste;

  liste.AddEnd( 3 );
  liste.AddEnd( 5 );
  liste.AddEnd( 7 );

  for( Liste<int>::Zugriff z = liste.Begin();
       z.IsValid(); z.Inc() )
  {
    std::cout << z.get() << std::endl;
  }

  return 0;
}
```

Der Typenparameter der Liste kann nun auch ein Zeiger sein. Damit kann sie auch ein Verhalten implementieren, wie die reine OO-Implementierung, die in Abschnitt 3.3 gezeigt wurde.

In gleicher Weise wie auch die Liste mit einem Inhaltstypen parametrisiert wurde, kann man es auch mit dem Vektor tun:

Listing 4.53. Eine Templateimplementierung des Vektors

```
template <typename T>
class Vektor
{
  typedef T ELEMENT;
public:
  Vektor()
  : size(0),
    capacity(STANDARDCAPACITY),
    delta(STANDARDDELTA)
    { Init();}
  Vektor( unsigned k = 10, unsigned d = 10 )
  : size(0),
    capacity(k>0?k:1),
    delta(d>0?d:1)
    { Init();}

  ~Vektor() { delete [] buf; }

  void AddEnd( ELEMENT e )
  {
    if( size == capacity )
      Grow();
    buf[size++] = e;
  }

  const ELEMENT& GetAt( unsigned idx ) const
  {
    return buf[idx];
  }
  const ELEMENT& operator[]( unsigned idx ) const
  {
    return buf[idx];
  }
  ELEMENT& GetAt( unsigned idx )
  {
    return buf[idx];
  }
```

```
  ELEMENT& operator[]( unsigned idx )
  {
    return buf[idx];
  }

  unsigned Size() const { return size; }

private:
  unsigned size;
  unsigned capacity;
  unsigned delta;
  ELEMENT *buf;
  enum { STANDARDCAPACITY = 10 };
  enum { STANDARDDELTA = 10      };
  void Init() { buf = new ELEMENT[ capacity ]; }
  void Grow();
};

template <typename T>
void Vektor<T>::Grow()
{
  unsigned newcapacity = capacity + delta;
  ELEMENT *newbuf = new ELEMENT[newcapacity];
  for( unsigned i = 0; i < size; ++i )
    newbuf[i] = buf[i];
  delete [] buf;
  buf = newbuf;
  capacity = newcapacity;
}
```

In diesem Abschnitt wurden die internen Repräsentationen von den Containern Liste und Vektor dargestellt. Im Abschnitt 5.2.6 über die STL auf Seite 292 werden die Container vorgestellt, die sich in der Standardbibliothek von C++ befinden. Dazu gehört auch eine Liste und ein Vektor. Was allerdings in dem genannten Abschnitt genauer behandelt wird, ist die Vereinheitlichung der Schnittstellen der Container.

4.1.15 Smart Pointer mit Templates

Eine Möglichkeit, Beziehungen zwischen Klassen abzusichern, stellen die Smart Pointer dar. Smart Pointer sind Klassen, die sich verhalten, als seien sie Zeiger, dafür aber etwas mehr Logik enthalten. Mit Smart Pointern lässt

sich beispielsweise ein Verhalten realisieren, dass der Garbage Collection in Java oder C# sehr ähnlich ist. In Java und C# werden alle Objekte dynamisch mit `new` erzeugt. Zerstören kann man die Objekte nicht. Wenn man sie nicht mehr braucht, verliert man einfach die letzte Referenz auf sie, womit sie an die Garbage Collection gehen. Diese wiederum entscheidet nach ihren Kriterien, wann und ob ein Objekt gelöscht werden soll. Das Verhalten mit den Referenzen ist sehr bequem. Der Programmierer braucht sich keine Gedanken über den Speicher zu machen, er „vergisst" einfach seine Objekte und der Rest wird von der Garbage Collection übernommen. Natürlich ist die Garbage Collection selbst zu aufwändig, um sie hier in C++ nachzuimplementieren. Das Verhalten mit den Referenzen allerdings lässt sich in C++ einfach realisieren.

Dazu muss eine Klasse modelliert werden, deren Objekte als Zeiger auf einen anderen Klassentyp fungieren können. Es müssen mehrere Zeiger auf ein Objekt verweisen können und beim Löschen des letzten Zeigers muss auch das abhängige Objekt zerstört werden. Überladene Operatoren können für den Zugriff auf den Inhalt sorgen.

Listing 4.54. Ein Template-basierter Smart Pointer

```
template <typename T>
class SmartPointer
{
public:
  SmartPointer( T *const ptr ) : pObj(ptr) {}
  ~SmartPointer() { delete pObj; }

  // Zugriffe
  T * operator->() { return  pObj; }
  T & operator*()  { return *pObj; }
private:
  T *pObj;
};
```

So ist die Klasse `SmartPointer` natürlich noch nicht vollständig. Um festzustellen, wann der letzte Zeiger auf ein Objekt gelöscht wurde, muss eine Referenzzählung durchgeführt werden. Für jedes referenzierte Objekt muss ein Referenzzähler existieren. Man könnte dafür einen einfachen `unsigned int`-Datentyp verwenden. An dieser Stelle soll aber eine Klasse `RefCounter` eingeführt werden, damit deutlich wird, welche Rolle die Objekte des Typs spielen. Außerdem lassen sich mit Inlinemethoden Optimierungsmöglichkeiten einbauen, die die Klasse ebenso effektiv machen wie den `unsigned int`-Datentyp selbst.

Listing 4.55. Ein Referenzzähler

```
class RefCounter
{
public:
    RefCounter() : n(0) {}
    void Inc() { n++; }
    void Dec() { n--; }
    bool IsZero() const { return n == 0; }
private:
    unsigned int n;
// Copy-Konstruktor und Zuweisungsoperator verstecken!
    RefCounter( const RefCounter & );
    RefCounter &operator=( const RefCounter & );
};
```

Nun kann die Klasse `SmartPointer` um den Referenzzähler erweitert werden. Dazu müssen der schon definierte Konstruktor und der Destruktor angepasst werden und Kopierkonstruktor und Zuweisungsoperatoren definiert werden.

Listing 4.56. Implementierung des Smart Pointers

```
template <typename T>
class SmartPointer
{
public:
  SmartPointer( T *const ptr )
  : pObj(ptr), rc( (ptr!=0) ? new RefCounter() : 0 )
  { if(rc)nrc->Inc(); }
  ~SmartPointer()
  {
    if(pObj)
    {
      rc->Dec();
      if( rc->IsZero() )
      {
        delete rc;
        delete pObj;
      }
    }
  }

  // Kopieren und zuweisen sichern
  SmartPointer(const SmartPointer &sp);
  const SmartPointer& operator=(const SmartPointer &sp);
  const SmartPointer& operator=(T *const ptr);
```

```
  // ...

private:
  T *pObj;
  RefCounter *rc;
};

template <typename T>
SmartPointer<T>::SmartPointer(const SmartPointer<T> &sp)
: rc(sp.rc)
, pObj(sp.pObj)
{
    rc->Inc();
}

template <typename T>
const SmartPointer<T>&
SmartPointer<T>::operator=(const SmartPointer<T> &sp)
{
    if( pObj == sp.pObj ) return *this;

    if( pObj )
    {
      rc->Dec();
      if( rc->IsZero() )
      {
        delete rc;
        delete pObj;
      }
    }

    rc = sp.rc;
    pObj = sp.pObj;
    rc->Inc();
    return sp;
}

template <typename T>
const SmartPointer<T>&
SmartPointer<T>::operator=(T *const p)
{
    if( p == pObj ) return *this;
```

```
    if( pObj )
    {
      rc->Dec();
      if( rc->IsZero() )
      {
        delete rc;
        delete pObj;
      }
    }

    rc = new RefCounter();
    rc->Inc();
    pObj = p;
    return *this;
}
```

Herkömmliche Zeiger können verglichen und auf Gültigkeit überprüft werden. Auch diese Eigenschaften können hinzudefiniert werden. Für die Vergleiche können Operatoren als `friend`-Deklarationen überladen werden[13]. Der Code bleibt damit auch zu älteren Compilern portabel, da keine globale Variante der Vergleichsoperatoren gewählt wurde, sondern die `friend`-Variante, die durch einen Nebeneffekt[14] erreichbar ist (siehe Abschnitt 4.1.11 auf Seite 242). Die Gültigkeitsüberprüfung lässt sich als Typenkonvertierungsoperator nach `bool` realisieren.

Listing 4.57. Implementierung von Vergleichsoperatoren für den Smart Pointer

```
template <typename T>
class SmartPointer
{
public:
  // ...
  friend
  inline bool operator==( const SmartPointer &p1,
                          const SmartPointer &p2 )
  {
    return p1.pObj == p2.pObj;
  }
  friend
  inline bool operator!=( const SmartPointer &p1,
                          const SmartPointer &p2 )
  {
```

[13] Würde man sie als Klassenelemente definieren, wären sie auch für eventuelle Kindklassen verfügbar.

[14] Über das Argument Dependent Lookup, oder das „Koenig-Lookup".

```
    return p1.pObj != p2.pObj;
  }
  operator bool() const { return pObj!=0; }
  // ...
};
```

Der folgende Testcode zeigt, wie der Smart Pointer angewendet werden kann.

Listing 4.58. Der Test der Smart-Pointer-Klasse

```
#include <list>
#include <cstdio>

class X
{
public:
    X()  { std::printf("X::X()\n"); }
    ~X() { std::printf("X::~X()\n"); }

    void f() { std::printf("X::f()\n"); }
private:
    X( const X& ); // verstecken!
};

void func( SmartPointer<X> p )
{
    p->f();
}

int main()
{
    SmartPointer<X> p1 = new X();
    SmartPointer<X> p2 = p1;
    SmartPointer<X> p3 = new X();
    SmartPointer<X> p4 = new X();

    p1->f();
    (*p2).f();
    p1 = p3; // Das erste Objekt wird nicht gelöscht.
    p4 = p3; // Das dritte Objekt wird gelöscht.
    func( p1 );

    return 0;
}
```

Die Ausgabe des Programms:

```
X::X()
X::X()
X::X()
X::f()
X::f()
X::~X()
X::f()
X::~X()
X::~X()
```

Ein solcher Smart Pointer kann auch als Templateargument der Liste übergeben werden. In diesem Fall löscht sie ihre enthaltenen Objekte selbst.

Listing 4.59. Smart Pointer in einem Container

```
// ...
typedef Liste<SmartPointer<X> > Container;

int main()
{
  Container l;
  l.AddEnd( new X() );
  l.AddEnd( new X() );

  return 0;
}
```

Ein solcher Smart Pointer kann viel zu einem stabilen Umgang mit dem Speicher in einer Anwendung beitragen. Natürlich kann man ihn nicht überall verwenden. Aber es gibt Situationen, in denen man Objekte dynamisch instanziieren muss. Objekte, die später nach einem aufwändigen Verfahren erhalten und gelöscht werden müssen. In diesen Situationen ist dieser Smartpointer eine große Hilfe. Dabei lässt sich die Implementierung auch weiterdenken. Wenn man den Smart Pointer Thread-sicher programmiert, kann er ein ausgezeichnetes Mittel zur Kommunikation zwischen Threads sein. Er kann das Command-Pattern erweitern helfen.

4.1.16 Projektorganisation mit Templates

Der praktische Einsatz von Templates in einem Projekt kann eine Hand voll Probleme mit sich bringen, für die es spezielle (Teil-)Lösungen gibt. Um an die Probleme heranzukommen, soll hier der Einsatz eines Templates in einem Projekt durchgespielt werden. Das folgende Beispiel zeigt die unvollständige Implementierung eines Vektors:

Listing 4.60. Teilimplementierung eines Template-basierten Vektors

```
template <typename T>
class Vector
{
public:
  Vector(int c = 50, int d = 50);
  ~Vector() { delete [] buf; }
  // ...
  void add( const T& );
  // ...
private:
  T *buf;
  int size;
  int capacity;
  int delta;
};

template <typename T>
Vector<T>::Vector( int c, int d )
: capacity(c), delta(d), size(0)
{
  if( this->delta <= 0 ) this->delta = 1;
  this->buf = new T[c];
}

template <typename T>
void Vector<T>::add( const T& t )
{
  if( this->size == this->capacity )
  {
    int newcapacity = this->capacity + this->delta;
    T* newbuf = new T[newcapacity];
    for( int i = 0; i < this->capacity; i++ )
      newbuf[i] = this->buf[i];
    delete [] this->buf;
    this->buf = newbuf;
    this->capacity = newcapacity;
  }
  this->buf[this->size++] = t;
}
```

Von dieser Vektorimplementierung sind in dem Beispiel nur der Konstruktor, der Destruktor und die `add()`-Methode implementiert. Der Destruktor ist implizit inline, da er in der Klasse definiert wurde. Wenn der Vektor in

verschiedenen Modulen verwendet werden soll, muss er in einer Headerdatei deklariert werden. Trennt man allerdings die Definition von der Deklaration, hat man das Problem, dass der Compiler die Definitionen der Vektormethoden nicht instanziieren kann, denn er kennt ja die Definitionen nicht, wo er sie verwenden muss. In der Datei der Templateimplementierung weiß er nichts über die Instanziierungen in anderen Modulen, außer das Compiler-Linker-Gespann hat einige Eigenschaften, die über die traditionelle Arbeitsteilung von Compiler und Linker in C hinausgehen.

Linkerprobleme

Das angesprochene Problem äußert sich dadurch, dass der Linker nicht vorhandene Symbole in den Objektdateien anmahnt. Wenn eine Funktion, eine Methode oder ein statisches Element beim Linken nicht gefunden wird, bricht der Linker mit einer Fehlermeldung ab. Bei der Instanziierung muss die Implementierung also bekannt sein, oder es müssen in einem separaten Modul explizite Instanziierungen mit dem gewünschten Parametertypen durchgeführt werden – mehr dazu im Abschnitt zur expliziten Instanziierung. Gehen wir also einmal davon aus, dass um eine korrekte Instanziierung der Templates aus dem vorangegangenen Listing zu gewährleisten, die vollständige Implementierung in der Headerdatei des Vektors vorhanden sein muss. Er besitzt demnach keine `.cpp`-Implementierungsdatei. Das heißt jedoch auch, dass in jedem Modul, in das diese Headerdatei aufgenommen wird, potentiell gleiche Templateinstanziierungen vorgenommen werden können. Das bedeutet wiederum, dass die entsprechenden Symbole mehr als nur einmal in den Objektdateien eines Projektes auftauchen können. Manche Linker haben auch damit ein Problem. Das kann sich in einer Fehlermeldung bemerkbar machen, in der der Linker mehrfach definierte Symbole beklagt. Das kann sich aber auch dadurch bemerkbar machen, dass die Codegröße des Endergebnisses eines Buildprozesses steigt. Nur wenige Linker sind heute in der Lage überflüssige Symbole aus dem Objektcode herauszuoptimieren. In Projekten, in denen es auf minimalen Footprint der Applikation oder des Systems ankommt, ist das beschriebene Problem wesentlich. Projekte, in denen man Compilezeiten verringern möchte, sind ebenfalls betroffen.

Inklusionsmodell

Das Inklusionsmodell beschreibt die Vorgehensweise, dass die vollständige Implementierung von Templates in den Modulen bekannt sein muss, in denen Instanziierungen der Templates durchgeführt werden. Im Inklusionsmodell werden also die Implementierungen von Templates in Headerdateien aufgenommen. Dabei können in Projekten die beschriebenen Linkerprobleme auftauchen. Ob sie zu Buche schlagen, hängt vor allem davon ab, welche Fähigkeiten das eingesetzte Compiler-Linker-Gespann mitbringt.

Compilezeitverhalten

Betrachtet man das Inklusionsmodell von der Seite des Compilezeitverhaltens, so muss man folgende Voraussetzungen betrachten: Erstens sind für die Implementierung der Methoden der Klassentemplates weitere Typen, d. h. weitere Headerdateien notwendig, die inkludiert werden müssen. Zweitens werden Instanziierungen eventuell mehrfach in unabhängigen Modulen durchgeführt. Beides, vor allem die erhöhte Abhängigkeit zu weiteren Headerdateien, die alle wiederholt compiliert werden müssen, führt zu einem enormen Anwachsen der Compilierzeit. Es hängt ganz vom Projekt ab, insbesondere von der Anzahl der Module, ob das Inklusionsmodell verwendet werden kann oder nicht. In großen Projekten wird oft ein Verfahren gewählt, das eine Trennung von Deklaration und Definition vorsieht, und für die Instanziierungen der Implementierungen eine separate Datei bereitstellt. Dieses Verfahren wird im folgenden Abschnitt erläutert.

Explizite Instanziierung

Es gibt eine Syntax zur expliziten Instanziierung von Templates. Diese Syntax sieht folgendermaßen aus:

Das Klassentemplate

Listing 4.61. Allgemeines Klassentemplate

```
template <typename T>
class X
{
public:
  void f();
};

template <typename T>
void X<T>::f()
{
}
```

kann explizit instanziiert werden durch:

Listing 4.62. Explizite Instanziierung einer Templateklasse

```
template class X<int>;
```

Nach dem Standard kann auch die Methode `X<T>::f()` instanziiert werden:

Listing 4.63. Explizite Instanziierung einer Template-Methode

```
template void X<int>::f();
```

Beim Instanziieren von Methoden- oder Funktionstemplates fehlt das Schlüsselwort `class` hinter `template`. Dieses Schlüsselwort deutet hier immer das explizite Instanziieren von Klassentemplates an. Es gibt noch die Regel, dass Templates nie zweimal explizit instanziiert werden dürfen. Auch dann nicht, wenn sie im Zuge einer Instanziierung eines umfassenden Elementes mit instanziiert wurden. Der folgende Code ist also ungültig:

Listing 4.64. Ungültige doppelte Instanziierung

```
template void X<int>::f();
template class X<int>; // Fehler!
```

Die Methode `void X<int>f()` wurde schon instanziiert und soll dann noch einmal im Zuge der Klasseninstanziierung angelegt werden. Das ist nach dem Standard nicht erlaubt.

Zu dem Thema muss aber auch gesagt werden, dass manche Compiler die explizite Instanziierung von Klassenmethoden nicht erlauben. Das Feature kann also nur dann eingesetzt werden, wenn es der Compiler auch unterstützt.

Ergänzung des Inklusionsmodells

Um die bschriebenen Nachteile des Inklusionsmodells zu minimieren, gibt es eine Strategie, die auf die explizite Instanziierung aufbaut. Dazu werden die Deklarationen und die Definitionen von Templates getrennt in zwei Headerdateien aufgenommen. Die Definition:

Listing 4.65. Reine Deklarations-Templates

```
// Verctor.h
template <typename T>
class Vector
{
public:
  Vector(int c = 50, int d = 50);
  ~Vector() { delete [] buf; }
  // ...
  void add( const T& );
  // ...
private:
  T *buf;
  int size;
  int capacity;
  int delta;
};
```

Die Deklaration:

Listing 4.66. Implementierungs-Templates

```
// VectorImpl.h
#include "Vector.h"

template <typename T>
Vector<T>::Vector( int c, int d )
: capacity(c), delta(d), size(0)
{
  if( this->delta <= 0 ) this->delta = 1;
  this->buf = new T[c];
}

template <typename T>
void Vector<T>::add( const T& t )
{
  if( this->size == this->capacity )
  {
    int newcapacity = this->capacity + this->delta;
    T* newbuf = new T[newcapacity];
    for( int i = 0; i < this->capacity; i++ )
      newbuf[i] = this->buf[i];
    delete [] this->buf;
    this->buf = newbuf;
    this->capacity = newcapacity;
  }
  this->buf[this->size++] = t;
}
```

In den Modulen, in denen Instanzen von `Vector` gebraucht werden, inkludiert man nur dir Datei `Vector.h`:

Listing 4.67. Instanziierung und Verwendung der Typeninformation

```
#include "Vector.h"

int main()
{
  // Hier wird nur die Typeninformation instanziiert.
  Vetor<int> v();

  // ...

  return 0;
}
```

Allerdings braucht man nun noch eine Datei, die als Instanziierungsmodul für das Projekt agiert. Nennen wir sie einmal `TemplateInst.cpp`:

Listing 4.68. Instanziierung der Implementierungen

```
// TeplateInst.cpp
#include "Vector.h"
#include "VectorImpl.h"

// Hier folgen die Instanziierungen der Templates,
// die im Projekt gebraucht werden.
template class Vector<int>;
template class Vector<double>;

// ...
```

Diese Datei inkludiert die beiden Headerdateien für das Klassentemplate `Vector<T>` und nimmt im folgenden Abschnitt alle expliziten Instanziierungen auf, die im Projekt benötigt wird. Das bedeutet nun, dass in allen anderen Modulen nur die Typeninformation der Klassentemplates instanziiert wird. In der Datei `TemplateInst.cpp` werden auch die Implementierungen instanziiert. Der Wermutstropfen an der Technik ist, dass man daran denken muss, die expliziten Instanziierungen in die Datei aufzunehmen, die an irgendeiner beliebigen anderen Stelle im Projekt anfallen.

Man gewinnt dabei ein deutlich günstigeres Compilezeitverhalten des Buildprozesses. Es werden nur gerade so viele Methoden und Funktionen im Objektcode angelegt, wie auch im Projekt gebraucht werden. Das heißt, dass man mit der Methode auch das Linkerproblem beseitigen kann. Schade ist nur, dass für die STL[15] eine solche Vorgehensweise nicht standardmäßig vorgegeben ist.

Das Separationsmodell

Für das Separationsmodell wurde ein eigenes Schlüsselwort `export` definiert. Im Grunde genommen geht es nur darum, das, was im vorigen Abschnitt beschrieben wurde, besser durch den Compiler unterstützen zu lassen. Der Compiler soll mit einer Zusatztechnik ausgestattet die expliziten Instanziierungen selbst vornehmen, d. h. die Instanziierungen werden dann trotz Trennung der Templatedeklaration von der Templatedefinition implizit durchgeführt. Dazu muss an der Deklaration das Schlüsselwort `export` angebracht werden.

[15] STL = Standard Template Library. Sie ist Bestandteil der C++-Standardbibliothek und enthält eine auf Templates basierende Containerbibliothek. Siehe Abschnitt 5.2.6 auf Seite 292.

Listing 4.69. Verwendung des Schlüsselworts `export`

```
// Verctor.h
export template <typename T>
class Vector
{
public:
  Vector(int c = 50, int d = 50);
  ~Vector() { delete [] buf; }
  // ...
  void add( const T& );
  // ...
private:
  T *buf;
  int size;
  int capacity;
  int delta;
};
```

Die Implementierungsdatei bleibt wie sie ist. Alle abhängigen Templates, wie z. B. die Methodentemplates für den Konstruktor und die Methode `f()`, werden damit auch „exportiert". Eine Instanziierungsdatei (eine extra Quellcodedatei, in der man alle Templateinstanziierungen aufnimmt) ist für das Projekt nicht mehr nötig. Genau das ist der Teil, den der Compiler nun für den Programmierer übernimmt. Man kann sich vorstellen, dass dafür eine ganz neue Technik notwendig ist, die es ermöglicht, Informationen zwischen den Übersetzungseinheiten auszutauschen. Das geht beispielsweise über eine separate Datei, in die der Compiler hineinschreibt, welche Templates instanziiert werden müssen und in welchen Dateien sich die Definitionen der Templates befinden. Eine solche Datei kann je nach Projekt sehr groß werden und den Übersetzungsvorgang zusätzlich verlangsamen.

Leider unterstützen die heutigen Compiler dieses Feature noch nicht, obwohl es im Standard steht. Mir ist gerade ein Compiler bekannt, der heute das Schlüsselwort `export` unterstützt: der Comeau C++-Compiler.

4.2 Konzepte für den Einsatz von Templates

Zunächst einige Grundstrukturen, um Metaprogrammierung[16] durchzuführen. Wenn zur Compilezeit etwas angepasst oder generiert werden soll, muss es Strukturen geben, die Entscheidungen durchführen, und (Compilezeit-) „Daten", die abgefragt oder generiert werden können. Um zu zeigen, dass die-

[16] Metaprogrammierung bezeichnet die Ausnutzung der generischen und generativen Fähigkeiten des Compilers.

se Dinge grundsätzlich möglich sind, sollen hier Ablaufkontrollen zur Compilezeit vorgestellt werden, wie sie in [CzarneckiEisenecker00] beschrieben sind. Dazu sollen die „Typetraits“ aus [Alexandrescu01] erarbeitet werden, damit ersichtlich wird, was z. B. zur Compilezeit abgefragt werden kann.

Eine Ablaufkontrolle zur Compilezeit, analog zur `if`-Anweisung während der Laufzeit eines Programms.

Listing 4.70. Compilezeitbedingungen mit Templates

```
template <bool COND, typename A, typename B>
struct IF
{
  typedef A RETURN;
};
template <typename A, typename B>
struct IF<false, A, B>
{
  typedef B RETURN;
};
```

Da zur Compilezeit Typen bearbeitet werden, sind Typen die „Funktionen der Compilezeit“. Abgefragt werden kann fast alles, was zu diesem Zeitpunkt schon feststeht: Konstanten und Typinformation. Eine Konstante wurde im vorherigen Beispiel schon abgefragt. Das folgende soll zeigen, wie man Typinformation abfragt.

Listing 4.71. Typendetektion zur Compilezeit

```
template <typename T>
struct IsPointer
{
  enum { VALUE = false };
};
template <typename T>
struct IsPointer<T*>
{
  enum { VALUE = true };
};
```

Andere Informationen zur Compilezeit können erzeugt bzw. verarbeitet werden.

Listing 4.72. Rechenanweisungen für den Compiler

```
template <int X, int Y>
struct POWER
{
  enum { VALUE = X * POWER<X,Y-1>::VALUE };
```

```
};
template <int X>
struct POWER< X, 0 >
{
  enum { VALUE = 1 };
};
```

In den vorangegangenen Beispielen wurde gezeigt, dass der Compiler dazu verwendet werden kann, Entscheidungen zu fällen, Typen zu detektieren und Rechenanweisungen auszuführen. In den folgenden Abschnitten werden Methoden des Einsatzes von Templates gezeigt.

4.2.1 „Policy Based Design"

Der Begriff des „Policy Based Design" wurde von Andrei Alexandrescu in [Alexandrescu01] erstmals eingeführt. Ein weiteres wichtiges Grundlagenwerk ist [CzarneckiEisenecker00]. Mit diesem Begriff wird ein aspektorientierter Programmieransatz bezeichnet, dessen Aspektbindung zur Compilezeit stattfindet[17]. Die Sprachmerkmale, mit denen dieses Policy Based Design verwirklicht wird, sind die Templates. Der Grundgedanke dieses Ansatzes liegt in der Dekomposition von Problemen in unabhängige Problemaspekte. Diese Problemaspekte werden unabhängig voneinander modelliert und mittels bestimmter Konstrukte zur Compilezeit zusammengeführt und in einen gemeinsamen Kontext gebracht. Damit die Zusammenführung realisiert werden kann, muss gegen Schnittstellen programmiert werden, die jedoch nicht mit der objektorientierten Technik der abstrakten Klassen gebildet werden. Es wird eine formale Vereinbarung über den Aufbau generischer Schnittstellen getroffen, die nicht darin besteht, dass eine Schnittstelle im Voraus deklariert wird. Die Schnittstelle muss einer Form entsprechen, die beim Übersetzungsvorgang als solchem überprüft wird. Dieser Übersetzungsvorgang determiniert die Schnittstelle im eigentlichen Sinne. Eine Schnittstellenbeschreibung gibt es also nicht in C++ selbst. Die Schnittstelle muss in Form einer Beschreibung für den Programmierer vorliegen. Dieser muss sich an bestimmte Vorgaben halten, die später allerdings vom Compiler überprüft werden können. Diese Vorgaben können zum Beispiel etwas darüber aussagen, welche Klassenelemente vorhanden sein müssen oder welche Parameter eine Methode braucht. Die Vorgaben können aber auch fordern, dass bestimmte Typen deklariert werden. Da es in dieser Technik keine Schnittstellenbeschreibungen in C++ selbst gibt, sind die Module stärker voneinander entkoppelt als bei der Verwendung von OO-Techniken.

[17] Siehe Abschnitt 4.3 auf Seite 273.

Die Dekomposition

Die im Policy Based Design durchgeführte Dekomposition gleicht im Wesentlichen der Dekomposition in anderen Entwicklungsmethoden. Auch in der Objektorientierten Programmierung finden sich Vorgehensweisen, die eine Dekomposition in sehr ähnlicher Weise ermöglichen, wie sie im Policy Based Design durchgeführt werden muss. Man unterteilt ein Problem in Teilprobleme, um diese besser lösen zu können. Im Abschnitt 4.3 ab Seite 274 wird die Dekomposition des Problemfelds „String" in der Standardbibliothek gezeigt. Dieser wird zur Compilezeit erst zu der Klasse zusammengebunden, die man aus der Standardbibliothek kennt. Die Teile, aus denen sich die String-Klasse zusammensetzt, definieren alphabetische Sortierregeln, Speicherverwaltung und Datenhaltung in der Weise eines Vektors. Diese Teile sind so geschrieben, dass sie zusammengefasst werden können. Dazu wurde eine Templatetechnik eingesetzt, die ohne explizite Schnittstellenbeschreibung in C++ auskommt.

Das Strategiemuster beispielsweise, das in dem Buch [GoF95] beschrieben ist, geht von Zielrichtung und Ausführung in die gleiche Richtung – nur eben mit objektorientierten Mitteln. Das auf der Templatetechnologie basierende Policy Based Design bildet die Dekomposition aber deutlicher im Code ab, als es objektorientierte Techniken vermögen. In den objektorientierten Methoden müssen immer noch Elternklassen eingebunden werden, was die Abhängigkeiten zwischen Modulen festigt.

4.2.2 Idiome für die Template - Programmierung

Die vielfältigen Ausdrucksmöglichkeiten von C++ stehen im Zusammenhang mit den verschiedenen Leitideen, die von dieser Sprache verfolgt werden. Allen voran ist die älteste und am besten theoretisch fundierte Leitidee, das Paradigma der Objektorientierten Programmierung. Die Objektorientierte Programmierung wird schon seit gut zwanzig Jahren mit C++ und anderen Sprachen durchgeführt. Die lange Arbeit mit dieser Methode führte dazu, dass sie eine gewisse Reife erreichte. Heute sind weniger grundsätzliche Fragen offen als beim Beginn der praktischen Anwendung der Objektorientierung in der Softwareentwicklung. Ein Entwickler ist nicht gut beraten, wenn er die gesamte Entwicklung, die vor seinem Einstieg in die Softwareentwicklun stattgefunden hat, selbst nachholen und erleben möchte. Wie in anderen Bereichen auch kann man auf Erfahrungen der Vorangegangenen aufbauen und sich die Arbeit dadurch erleichtern. Aber in welcher Form liegen diese Erfahrungen vor? Und wie werden sie mitgeteilt? Die Antwort auf diese beiden Fragen hat der Schweizer Informatiker Erich Gamma in seiner Publikation „Entwurfsmuster oder Elemente wiederverwendbarer objektorientierter Software" geliefert, die 1994 zum ersten Mal erschienen ist. Dieses Buch revolutionierte die

Objektorientierte Programmierung. Gamma lieferte mit seinem Buch einen Katalog von musterhaften Lösungen für verschiedene, immer wiederkehrende Situationen in der objektorientierten Entwicklung. Die Lösungen waren dabei alle nicht neu. Die Leistung Gammas bestand darin, die Lösungen unter allen vorhandenen gut ausgewählt, abstrahiert und benannt zu haben. Die Namen dieser „Muster" helfen heute den Entwicklern über ihre Entwürfe zu sprechen. Sie sind die Begriffe des Softwareentwurfs. Vor 1994 fehlte das begriffliche Instrumentarium, um Softwarestrukturen adäquat zu beschreiben. Er führte also mit seinen 23 Mustern eine Sprache für den objektorientierten Softwareentwurf ein. Diese Begrifflichkeit ist unabhängig von der eingesetzten Sprache. Mit der Sprache Java wurde eine sehr stark musterorientierte Technik in den Bibliotheken realisiert. Die hohe Produktivität, die mit Java möglich ist, ist in der Hauptsache auf diese Musterbasiertheit zurückzuführen[18]. Dem steht der Entwurf von C++ infolge des höheren Alters natürlich nach. Ebenso ist die Standardbibliothek von C++ nicht so einheitlich strukturiert und bedient sich weit weniger aus dem Mustergedanken. Umso mehr muss der C++-Entwickler Wissen über Muster aufbauen. Schließlich wird er durch Bibliothek und Sprache nicht dazu gezwungen, schon getestete Strukturen zu implementieren. Er muss sie kennen und selbst neu aufbauen, wenn er wartbaren Code in kurzer Zeit schreiben möchte. Natürlich hat sich die Anzahl der Muster durch verschiedene Autoren über die Zeit noch vergrößert. Dabei stellen die Muster von Gamma einen Grundstein der Mustersprachen dar, die sich entwickelten.

Für die Templateprogrammierung hat sich noch keine Mustersprache im Sinne der OO-Entwurfsmuster gebildet. Ansätze finden sich in [Alexandrescu01] und heißen beispielsweise „Type-Traits", „Type-Lists" oder „Policy-Design". Das Fehlen einer so mächtigen Mustersprache, wie sie für die Objektorientierte Programmierung existiert, hemmt noch den breiten Einsatz der Templatetechnik in der industriellen Softwareentwicklung.

4.2.3 Compiletime - Assertions

Templatetechnologien können auch dazu genutzt werden, zur Compilezeit Bedingungen zu überprüfen, die zu diesem Zeitpunkt schon feststehen. Das sind beispielsweise Compilerflags oder mit `#define` definierte Konstanten. So können zum Beispiel Versionsinformationen von Bibliotheken oder Komponenten in der Form von symbolischen Konstanten abgelegt werden. Es können auch Eigenschaften von Typen zur Compilezeit abgefragt werden. Dabei ergeben sich einige sinnvolle Anwendungsbereiche, die auch außerhalb der aspektorientierten Programmierung von Bedeutung sind. In diesem

[18] Java zwingt den Entwickler an vielen Stellen zur Einhaltung musterhafter Strukturen, ohne bei ihm das Wissen darüber vorauszusetzen.

Abschnitt sollen die Grundlagen für den Einsatz von Compilezeit-Assertions vorgestellt werden. Im Wesentlichen stammt diese Methode aus [Alexandrescu01].

Zunächst baut das Prinzip der Compilezeit-Assertion auf teilweise implementierte Templates auf:

Listing 4.73. Compilezeit-Assertion aus [Alexandrescu01]

```
template <bool> struct CompileTimeError;
template <> struct CompileTimeError<true> {};
#define CT_CHECK(expression) \
        (CompileTimeError<(expression) != 0>())
```

Wenn der Ausdruck zur Laufzeit den Wert `0` oder `false` ergibt, kann das Objekt der Templatestruktur nicht instanziiert werden, da keine Implementierung vorhanden ist. Damit kann der Code nicht übersetzt werden. Der Compiler gibt eine Fehlermeldung aus, in der er die fehlende Definition von `CompileTimeError<false>` anmahnt. Damit ist zumindest ansatzweise zum Ausdruck gebracht, dass es sich um einen Compilezeitfehler handeln könnte.

4.3 Aspektorientierte Programmierung

Der aspektorientierte Ansatz beschäftigt sich mit der Dekomposition von Problemstellungen in unabhängige Aspekte. Er definiert auch Techniken des Zusammenbindens von Aspekten zu unterschiedlichen Zeitpunkten. Wesentliche Zeitpunkte für die Aspektbindung sind Entwicklungszeit, Übersetzungszeit, Aufstartzeit und Laufzeit. Es gibt verschiedene Programmiersprachen und Erweiterungen zu bestehenden Sprachen, die ein Aspektbinden zu unterschiedlichen Zeiten erlauben. Wichtige Vertreter für OO-Sprachen mit aspektorientierten Ansätzen sind Java ab Version 1.5, AspectJ für Java und eben C++. Während die Sprache Java durch die Reflection-Eigenschaft grundsätzlich alle Zeitpunkte des Aspektbindens zulässt, ist in C++ nur das Aspektbinden zur Entwicklungs- und zur Übersetzungszeit möglich. Spätere Zeitpunkte können nur mit einem starken Eingriff in den Code emuliert werden und verkomplizieren den fachlichen Code mit Strukturen, die schwer zu warten und eventuell auch schwer auf neue Anforderungen erweiterbar sind.

Das syntaktische Mittel, mit dem Aspektorientierung in C++ erreichbar ist, ist die Templateprogrammierung. Mit den Templates lassen sich unabhängige Aspekte definieren, die sich durch den Compiler zu Klassen binden lassen. Grundlagentheorien für den aspektorientierten Ansatz finden sich in [CzarneckiEisenecker00] und in dem Buch von Alexandrescu [Alexandrescu01] mit dem Policy Design. In [GoF95] wird als eine mögliche Umsetzung des Strategiepatterns eine Implementierung mit Templates

genannt, die von Grund auf anders aufgebaut ist als die objektorientierte Variante. Das Strategiemuster kommt der Idee des Policy Designs sehr nahe. Schon die ersten Containerbibliotheken, die Anfang der neunziger Jahre mit den ersten C++-Entwicklungsumgebungen ausgeliefert wurden und auf Templates basierten, nutzten die Grundidee der Identifizierung unabhängiger Aspekte und der Dekomposition mit Hilfe dieser neuen Technik. Die Vereinheitlichung der unterschiedlichen Containerbibliotheken mit der Entwicklung der STL durch Stefanov brachte einen Durchbruch der Technik, wenngleich in einem sehr begrenzten Anwendungsbereich.

Weitere Teile der ANSI-C++-Standardbibliothek wurden nach den gleichen Prinzipien gestaltet, die schon seit einigen Jahren in der STL erfolgreich angewendet und getestet wurden.

Der Standardstring, eine erfolgreiche Dekomposition

Der String aus der C++-Standardbibliothek wird im Unterkapitel 5.2.7 auf Seite 326 vorgestellt. Es sollen hier nicht seine einzelnen Funktionen im Besonderen betrachtet werden, sondern die Art und Weise, wie er dekomponiert und in einzelne Aspekte getrennt wurde.

Hier noch einmal die Deklaration des Strings im Namespace `std`. Er basiert auf dem Template `basic_string`, das insgesamt drei Templateparameter hat. Die drei Templateparameter spiegeln die drei wesentlichen Aspekte einer Stringimplementierung wider.

Listing 4.74. Die Template-Parameter des Strings in der Standard-Library

```
template < class E,
           class T = char_traits<E>,
           class A = allocator<E> >
class basic_string;

//...

typedef basic_string<char> string;
```

Der erste Aspekt einer Stringimplementierung ist der atomare Basistyp, auf dem die Codierung des einzelnen Zeichens beruht. In der Vergangenheit wurden hauptsächlich acht-bittige[19] Zeichentypen verwendet um Codierungen wie ASCII oder UDF zu verwirklichen. Seit Anfang der neunziger Jahre ist das 16-bittige Unicode im Vormarsch. Der Vorteil eines 16-bittigen Grundtyps zur Codierung von Zeichen hat den Vorteil, dass viel mehr Zeichen dargestellt werden können. Mit 2^{16} möglichen unterschiedlichen Zeichen im Unicode gegenüber 2^8 dem Datentyp `char` kann man mehrere Alphabete in einem Zeichensatz definieren. Ein Text kann also Buchstaben unterschied-

[19] Auf sieben-bittige EBSDIC-Codierungen wird der Einfachheit halber nicht eingegangen.

licher Alphabete enthalten, ohne dass zwischen Zeichensätzen umgeschaltet werden müsste. C++ stellt dafür den Datentyp `wchar_t` zur Verfügung. Welcher Grundtyp für die Einzelzeichen gewählt wird, ist unabhängig von der sonstigen Logik, die die Stringklasse implementieren muss.

Der zweite Aspekt betrifft die Art und Weise, wie sich Zeichen und Zeichenketten alphnumerisch zueinander verhalten. Der Parameter, der in der vorangegangenen Deklaration mit `T` bezeichnet ist, stellt einen Typ dar, der Methoden transportiert, die genau diese Beziehungen regeln. Es gibt in diesem Typ beispielsweise Methoden, die definieren, in welcher Weise Zeichenketten miteinander verglichen werden. Dadurch wird eine alphabetische Reichenfolge festgelegt.

Der dritte Aspekt (Parameter `A`) betrifft eine Anforderung des Systems. Jede Zeichenkette braucht Platz im Speicher. Dieser Platz muss dynamisch allokiert werden, denn die benötigte Größe ergibt sich erst zur Laufzeit. Wenn Operationen mit Zeichenketten durchgeführt werden, wie zum Beispiel das Kopieren eines Strings oder das Anhängen an einen anderen, dann müssen sehr oft neue Bereiche allokiert und alte freigegeben werden. Manch ein System kann ein vielfältiges Allokieren und Deallokieren schlecht verkraften, wenn es etwa keine virtuelle Speicherverwaltung besitzt. In diesem Fall arbeitet man gerne mit vorallokierten Speicherbereichen um die Anzahl der Allokationen und Deallokationen in Grenzen zu halten. Dabei werden teilweise sehr trickreiche Strategien angewandt, um trotz der Vorallokierung einem zu großen Speicherverbrauch entgegenzuwirken. Es gibt auch die entgegengesetzten Strategien, die vor allem Geschwindigkeit zum Ziel haben. Welche Strategie auch immer mit einer aktuellen Stringklasse angewendet wird – sie ist immer in Form einer Allokatorklasse implementiert, die als Templateparameter `A` dem `basic_string` übergeben wird.

Die Schnittstellen der beiden Typen `T` und `A` müssen ganz bestimmten Vorgaben genügen, damit der Compiler diese beiden Templates im Template `basic_string` zu einer Einheit verschmelzen kann, die auch keine Laufzeitnachteile[20] mit sich bringt. Wie zu Beginn des Unterkapitels schon angesprochen wurde, wird damit durch den Compiler das Binden der Aspekte vollzogen.

[20] Es werden zum Beispiel keine virtuellen Methoden verwendet. Es sind ja alle Bedingungen zur Compilezeit bekannt und die Templates ermöglichen ein typsicheres Anpassen der Schnittstellen durch den Compiler. Es ist also keine Entscheidung zur Laufzeit mehr nötig.

5 Die C++ - Standardbibliothek

In den vorangegangenen Kapiteln wurden viele Prinzipen und Konzepte erklärt, die unter anderen in der Standardbibliothek Anwendung finden. Dort sind sie allerdings komprimiert und ergänzen sich gegenseitig. Der Schlüssel zu einer erfolgreichen Verwendung der Standardbibliothek besteht in der Kenntnis der zugrunde liegenden Techniken und natürlich auch aus einem Überblick über deren Elemente und Teilbereiche. In diesem Kapitel sollen deshalb die wesentlichen Elemente der Standardbibliothek vorgestellt werden. In der ersten Tabelle im Abschnitt 5.2 auf Seite 278 stehen alle Headerdateien, die zur Standardbibliothek gehören und deren Domäne. Für alle diese Teilbereiche finden sich in diesem Kapitel Beispiele.

5.1 Die Geschichte der Standardbibliothek

C++ basiert auf der Programmiersprache C und enthielt von Anfang an die Bibliothek der Vorgängersprache. In den AT&T-Standards wurden schon früh C++-spezifische Teile in die Bibliothek aufgenommen. So wurden zum Beispiel die C++-Streams schon sehr früh in den Bibliotheksstandard von AT&T integriert. Diese basierten damals allerdings viel stärker auf Polymorphismus als heute in der ANSI/ISO-Standardbibliothek. Der ANSI/ISO-Standard nutzt in seiner Bibliotheksimplementierung viel mehr die Templatetechnologien, die recht spät in die Sprache C++ aufgenommen wurden. Viele Hersteller von C++-Entwicklungswerkzeugen erkannten schon früh, dass für die Modellierung von 1-zu-n-Relationen Containerklassen nötig sind, und fügten ihren Bibliotheken herstellerspezifische Containerbibliotheken bei. Diese Container basierten meistens auf Polymorphismus und stellten Listen, Stacks, Vektoren und manchmal auch Binäre Bäume dar. Ab etwa 1994 wurde die STL als templatebasierte Containerbibliothek einer großen Zahl von C++-Entwicklern verfügbar. Sie war öffentlich zugänglich und zu dieser Zeit unterstützten schon viele wesentliche C++-Compiler die Templates. Sie entwickelte sich zum Quasistandard bei der Entwicklung portierbaren Codes. In manchen Bereichen, wie zum Beispiel der Entwicklung für das Betriebssystem Windows, setzte sich die STL nicht so gut durch, da dort schon sehr stark die MFC-Bibliothek[1] eingesetzt wurde. In der MFC sind Container enthalten, die auch ein wenig die Templatetechniken benutzen, darin aber weit

[1] Die Microsoft Foundation Classes. Ein Framework zur Windowsprogrammierung, das mit den Microsoft Visual C++-Compilern ausgeliefert wird.

hinter der STL zurückbleiben. Da die MFC selbst nicht portabel ist, gibt es in Projekten, die sie einsetzen, auch selten den Anreiz, in der Benutzung von Containern portabel zu sein. Neben der guten Portabilität der STL stach die exzellente Erweiterbarkeit und die saubere Modularisierung gegenüber anderen Containerbibliotheken hervor. Der dritte Aspekt, in dem die STL deutlich gegenüber anderen Bibliotheken punkten konnte, war die Berechenbarkeit der Kosten aller Operationen, die sie zur Verfügung stellt. Das alles führte dazu, dass sie für den ANSI-Standard von C++ vorgeschlagen und schließlich auch aufgenommen wurde. Seit 1998 ist die STL also in der Standardbibliothek von C++. Viele der Techniken, die sie benutzt, finden auch in anderen Bereichen der Standardbibliothek Anwendung. Mit der STL wurde sehr erfolgreich der Nutzen der Templates in C++ vorgeführt, der Anfang der neunziger Jahre noch sehr umstritten war. Seit 1998 gibt es fast keinen Bereich mehr in der Standardbibliothek, der ohne Templates auskommt.

5.2 Die Teilbereiche der Standardbibliothek

Die ANSI-C++-Standardbibliothek ist in einige Teilbereiche untergliedert. In der folgenden Tabelle werden alle Headerdateien der Bibliothek aufgelistet und einem der Bereiche zugeordnet. In den folgenden Abschnitten werden dann alle Bereiche durchgesprochen und in Beispielen vorgeführt. Dabei können nicht alle Elemente vollständig besprochen werden. Das würde den Umfang des Buches bei weitem sprengen. Hierzu sei vor allem auf [ISO-C++] verwiesen. Wichtig sind die hinter der Bibliothek stehenden Konzepte. In jedem der Teilbereiche finden sich besondere Konzepte, die ganz typisch für die Anwendung der Elemente sind. Wenn man die Bibliothek beherrschen möchte, muss man diese Konzepte beherrschen. Sie sind viel wichtiger als das auswendige Beherrschen von Teilen der Bibliothek.

Headerdatei	**Inhalte**
`<memory>` `<new>`	Speicherverwaltung

`<fstream>`
`<iomanip>`
`<ios>`
	Ein- und Ausgabestreams

`<iosfwd>`
`<iostream>`
`<istream>`
`<ostream>`
`<sstream>`
`<streambuf>`	

`<locale>`	Nationale Einstellungen

`<deque>`
`<list>`
`<map>`
`<queue>`
`<set>`
`<stack>`
`<vector>`
	Die Container der STL

`<iterator>`
`<algorithm>`
`<functional>`
	Iteratoren, Funktoren und Algorithmen der STL

`<string>`	Strings
`<exception>` `<stdexcept>`	Exception Handling
`<utility>`	Hilfsklassen

`<bitset>`
`<complex>`
`<limits>`
`<numerics>`
`<valarray>`
	Numerische Berechnungen und Matrizen

`<typeinfo>`	Typinformation

`<cassert>`	C++-Varianten der C-Headerdateien
`<cctype>`	
`<cerrno>`	
`<cfloat>`	
`<ciso646>`	
`<climits>`	
`<clocale>`	
`<cmath>`	
`<csetjmp>`	
`<csignal>`	
`<cstdarg>`	
`<cstddef>`	
`<cstdio>`	
`<cstdlib>`	
`<cstring>`	
`<ctime>`	
`<cwchar>`	
`<cwctype>`	

Die nachfolgenden Abschnitte gehen durch einige Teilbereiche der Bibliothek. Dabei werden die jeweils besonderen Technologien vorrangig behandelt. Es finden sich natürlich auch Angaben dazu, in welcher Headerdatei welche Elemente definiert sind.

5.2.1 Die Streams

Die Streams stellen neben den Containern der STL den größten Teilbereich der Standardbibliothek dar. Im Allgemeinen ist das Streamobjekt `cout` gut bekannt, das die Standardausgabe repräsentiert. Neben `cout` gibt es `cerr` und `clog` für Fehlerausgaben und `cin` für Eingaben.

Im Vergleich zu der Funktion `printf()` aus C sieht die Anwendung der Streamobjekte in C++ etwas eleganter aus:

Listing 5.1. Einfache Anwendung von C++-Streams

```
#include <iostream>
#include <fstream>
```

```
int main()
{
  std::cout << "Ein Text" << std::endl;

  std::ofstream datei( "test.txt" );
  datei << "Ein Text" << std::endl;

  return 0;
}
```

An vielen Stellen in diesem Buch wurde der Operator `<<` für Streams schon verwendet. In Listing 5.1 wird gezeigt, dass sich der Streamoperator `<<` auf verschiedenartige Streams anwenden lässt. Dieser Ausgabeoperator ist für viele verschiedene Typen überladen. Dazu zählen alle Standarddatentypen und auch einige Klassen aus der Standardbibliothek. Für selbstdefinierte Typen kann man den Operator selbst überladen. Wie das geht, wurde schon in Abschnitt 2.2.35 gezeigt. Dort wurde der Ausgabeoperator für den Typ `Complex` überladen:

Listing 5.2. Überladen des Shift-Operators zur Verwendung als Ausgabeoperator

```
struct Complex
{
  double r;
  double i;
};

std::ostream& operator<<( std::ostream &os,
                          const Complex &c )
{
  os << "(" << c.r << "/i" << c.i << ")";
  return os;
}
```

Der Ausgabeoperator für den Typ `Complex` lässt sich nun auf verschiedenartige Streams anwenden, wie in Listing 5.3 gezeigt wird:

Listing 5.3. Anwendung des definierten Ausgabeoperators auf verschiedene Streams

```
#include <iostream>
#include <fstream>

struct Complex
{
  double r;
  double i;
};
```

```
std::ostream& operator<<( std::ostream &os,
                  const Complex &c )
{
  os << "(" << c.r << "/i" << c.i << ")";
  return os;
}

int main()
{
  Complex c = { 5, 3 };

  std::cout << "Eine Komplexe Zahl: " << c << std::endl;

  std::ofstream datei( "test.txt" );
  datei << "Eine Komplexe Zahl: " << c << std::endl;

  return 0;
}
```

Dass sich der Ausgabeoperator für mehrere Ausgabe-Streamarten gleichzeitig definieren lässt, liegt daran, dass diese in einer Vererbungshierarchie von der Klasse `std::ostream` abgeleitet sind.

Ebenso wie die Ausgabeoperatoren lassen sich Eingabeoperatoren `>>` für Spezialisierungen von `std::istream` überladen. Bei diesen speziellen Operatoren, die von einem Eingabestream lesen, spricht man auch von *Extraktoren*.

Listing 5.4. Definition und Anwendung eines Eingabeoperators

```
#include <iostream>
#include <fstream>

struct Complex
{
  double r;
  double i;
};

std::istream& operator>>( std::istream &is, Complex &c )
{
  is >> c.r;
  is >> c.i;
  return is;
}
```

```
std::ostream& operator<<( std::ostream &os,
                  const Complex &c )
{
  return os << "(" << c.r << "/i" << c.i << ")";
}

int main()
{
  Complex c;

  std::cout << "Eingabe einer Komplexen Zahl: ";
  std::cin >> c;
  std::cout << "Die Komplexe Zahl: " << c << std::endl;

  return 0;
}
```

Wie in Abschnitt 2.2.35 schon besprochen wurde, sind die Rückgabewerte der Operatoren für deren Verkettbarkeit wichtig. Sowohl der Eingabe-, wie auch der Ausgabeoperator kann in einer Reihe eingesetzt werden. In einer solchen Reihe reicht ein ausgeführter Operator das Streamobjekt durch die Funktionsrückgabe an den nächsten Operator weiter, der auf seiner rechten Seite steht. Dieser nimmt das Streamobjekt als linken Operanden entgegen, führt seine Operation aus und reicht das Streamobjekt wieder weiter. Die Ein- und Ausgabeoperatoren für die Standardstreams werden durch den Applikationsentwickler immer als globale Funktionen und nie als Klassenelemente deklariert, da man die Streamklassen selbst nicht manipulieren sollte[2].

Natürlich haben die Streamobjekte auch Methoden, die eine weiter reichende Einflussmöglichkeit auf Ein- und Ausgabe ermöglichen. Die Eingabeoperatoren interpretieren beispielsweise Leerzeichen, Tabulatoren und Zeilenumbrüche als Trennzeichen zwischen den zu lesenden Abschnitten. So trennen die Eingabeoperatoren in Listing 5.4 die Eingaben der Real- und Imaginäranteile der Komplexen Zahl durch eines der genannten Zeichen. Möchte man Leerzeichen, Tabulatoren und Zeilenumbrüche auch einlesen, so ist man auf die Anwendung bestimmter Operatoren angewiesen.

Das Beispiel in Listing 5.5 kopiert alle Zeichen von der Standardeingabe in die Standardausgabe. Dabei werden Leerzeichen, Tabulatoren und Zeilenumbrüche aus dem Text gelöscht.

Listing 5.5. Whitespace löschende Wirkungsweise des Eingabeoperators

```
// Streambeispiel5.cpp
#include <iostream>
```

[2] Zur Erinnerung: Ein Operator kann immer nur als Klassenelement derjenigen Klasse definiert werden, die den Typ seines linken Operanden abgibt.

```
int main()
{
  char c;
  while( std::cin >> c )
    std::cout << c;

  return 0;
}
```

Der Aufruf des Programms in der folgenden Weise erzeugt eine etwas unansehnliche Ausgabe. `Streambeispiel5 < Streambeispiel5.cpp` erzeugt

```
  //Streambeispiel5.cpp#include<iostream>i
  ntmain(){charc;while(std::cin>>c)std::co
  ut<<c;return0;}
```

Eine kleine Modifikation an dem Programm gibt den Text mit allen Zeichen auf der Standardausgabe aus:

Listing 5.6. Anwendung der Methode `get()` zum Einlesen aller Zeichen

```
// Streambeispiel6.cpp
#include <iostream>

int main()
{
  char c;
  while( std::cin.get( c ) )
    std::cout << c;

  return 0;
}
```

Die Methode `get()` liest alle Zeichen aus dem Eingabestrom.

In Listing 5.5 und in Listing 5.6 wurde die lesende Operation einfach in die Bedingungsklammer eines `while`-Konstruktes aufgenommen. Dass das machbar ist, hat damit zu tun, dass sowohl der Eingabeoperator `>>` als auch die Methode `get()` eine Referenz auf den Eingabestream zurückliefern. Dieser Stream hat einen Typenkonvertierungsoperator nach `void *`, der entweder einen Zeiger auf das nächste Byte im Stream liefert, oder `NULL`, wenn das Ende der Eingabedatei erreicht ist.

Diesen Zustand eines Streamobjekts kann man auch mit Methoden abfragen. So hat jeder `istream` die Methode `eof()`, die das Ende der Eingabe signalisieren kann. Die Methode `good()` liefert `true`, wenn der Stream in einem konsistenten Zustand ist. Eine Operation auf den Stream könnte also Erfolg haben. Die Methode `fail()` liefert `true`, wenn eine folgende Operation nicht mehr mit Erfolg durchgeführt werden kann. Es gibt noch die Methode `fail()`, die `true` liefert, wenn der Stream überhaupt nicht mehr verwendbar ist.

5.2.2 Formatierungen auf einem `ostream`

Die Ausgaben auf einem `ostream` können formatiert werden. Dazu stehen Flags aus der Basisklasse `ios_base` zur Verfügung. Drei davon betreffen das Zahlensystem, in dem die auszugebenden Werte dargestellt werden sollen: `dec` für dezimale, `hex` für hexadezimale und `oct` für oktale Darstellung. Die Vorgabeeinstellung ist `dec`.

Listing 5.7. Formatierung einer Ausgabe auf ein `ostream`-Objekt über Flags

```
#include <iostream>

int main()
{
  using namespace std;
  int i = 42;

  ios_base::fmtflags old = cout.flags(ios_base::oct);
  cout << i << endl;
  cout.flags(ios_base::hex);
  cout << i << endl;
  cout.flags(old);
  cout << i << endl;

  return 0;
}
```

Das Programm in Listing 5.7 erzeugt die Ausgabe:

```
52
2a
42
```

Andere Flags regeln die Ausrichtung oder fügen Füllzeichen ein.

Listing 5.8. Definition einer Feldbreite in einer Ausgabe

```
#include <iostream>

int main()
{
  using namespace std;

  cout.flags(ios_base::right);

  for( int i = 1; i < 7; ++i )
```

```
  {
    cout.width(7);
    cout << (i*i*i) << endl;
  }

  return 0;
}
```

In Listing 5.8 wird mit dem Flag `right` eine rechtsseitige Ausrichtung und mit der Methode eine Feldbreite von sieben Zeichen eingestellt. Das Programm erzeugt die Ausgabe:

```
      1
      8
     27
     64
    125
    216
```

Es gibt eine große Anzahl von Möglichkeiten, Ausgabestreams mittels Methoden und Flags zu steuern. Im nächsten Abschnitt wird eine weitere Variante der Streammanipulation vorgestellt.

5.2.3 Die Manipulatoren

Die Standardbibliothek definiert Steuerelemente, die einem Stream über den Ausgabeoperator übergeben werden können und die bestimmte Einstellungen vornehmen. Diese Steuerelemente nennt man Manipulatoren. Im vorangegangenen Abschnitt haben wir Flags kennengelernt, die es ermöglichen ganzzahlige Ausgaben in einem veränderten Zahlensystem auszugeben oder Ausrichtungen festzulegen. Für all diese Flags gibt es vordefinierte Standardmanipulatoren. Das Programm in Listing 5.9 entspricht also ganz dem in Listing 5.7 und erzeugt die gleiche Ausgabe.

Listing 5.9. Formatierung über Manipulatoren

```
#include <iostream>

int main()
{
  using namespace std;
  int i = 42;

  cout << oct << i << endl;
  cout << hex << i << endl;
  cout << dec << i << endl;
```

```
  return 0;
}
```

Auch das Programm aus Listing 5.8 lässt sich mit Manipulatoren umformulieren. Allerdings haben wir in dem genannten Programm auch einen Methodenaufruf am Streamobjekt mit einer Parameterübergabe. Es muss also die Methode `width()` durch einen Manipulator ersetzt werden, der einen Parameter transportieren kann. Auch die Methoden, die Einstellungen an Streams vornehmen, können ebenso wie die Flags durch Manipulatoren ersetzt werden. Dazu benötigt man die Manipulatoren mit Argumenten, die in der Headerdatei `<iomanip>` stehen. Das folgende Beispiel in Listing 5.10 formt das Beispiel in Listing 5.8 um:

Listing 5.10. Ein Manipulator mit einem Parameter

```
#include <iostream>
#include <iomanip>

int main()
{
  using namespace std;

  cout << right;

  for( int i = 1; i < 7; ++i )
  {
    cout << setw(7) << (i*i*i) << endl;
  }

  return 0;
}
```

Mit dem Manipulator `setfill()` lassen sich die definierten Leerräume auffüllen. Ändert man also die Ausgabeoperation in der Schleife folgendermaßen ab:

Listing 5.11. Feldbreite und Leerraumformatierung mit Manipulatoren

```
// ...
cout << setw(7) << setfill('-') << (i*i*i) << endl;
// ...
```

dann erzeugt das Programm die Ausgabe:

```
------1
------8
-----27
-----64
```

```
----125
----216
```

Man kann Manipulatoren selbst definieren. Dabei muss man sich nur an die vorgegebene Form der Manipulatoren halten.

Die drei möglichen Formen sind:

```
ios &     manipulator(ios &);
istream & manipulator(istream &);
ostream & manipulator(ostream &);
```

In Listing 5.12 wird die Definition eines Manipulators anhand einer Feldformatierung demonstriert.

Listing 5.12. Definition eines eigenen Manipulators

```
#include <iostream>
#include <iomanip>

std::ostream & myfield(std::ostream &os)
{
  using namespace std;
  return os << setw(7) << setfill('-');
}

int main()
{
  using namespace std;

  cout << right;

  for( int i = 1; i < 7; ++i )
  {
    cout << myfield << (i*i*i) << endl;
  }

  return 0;
}
```

Mit den Manipulatoren in C++ lassen sich Streams in einer einfachen und effektiven Weise formatieren. Sie sind den Formatierungen und Einstellungen über Flags weit überlegen, da sich die Form ihrer Anwendung besser in das Konzept der Streamoperatoren einfügt. Da letztendlich alles auf Funktionsaufrufe zurückgeführt wird, sind sie nicht etwa schneller. Man kann sie durch geschicktes Inlining allerdings so gestalten, dass sie auch nicht langsamer sind.

5.2.4 Die File - Streams

In Listing 5.3 wurde schon eine Datei mit Hilfe der Streamklasse `ofstream` beschrieben. Grundsätzlich ist also die Anwendung von File-Streams ebenso unkompliziert wie die der Standardeingabe und der Standardausgabe. Man gibt den Konstruktoren der Klassen `ifstream` und `ofstream` einfach einen Dateinamen mit, und das Objekt ist als Stream verwendbar, wenn die Datei geöffnet werden konnte. Die beiden Klassen haben auch Standardkonstruktoren, die ein Objekt kreieren, das nicht mit einer Datei verbunden ist. Die Verbindung mit einer Datei kann dann mit der Methode `open()` hergestellt werden, die den Dateinamen erwartet.

Was man für ein File-Stream noch verändern kann, ist die Interpretation der eingelesenen Zeichen. So gibt das Flag `ios::binary` an, dass der Stream binär eingelesen werden soll. Der Unterschied zur standardmäßigen textuellen Lesevariante ist die, dass der Zeilenumbruch auf bestimmten Systemen aus zwei Zeichen besteht[3], auf anderen nur aus einem. Diese zwei Zeichen werden also zu einem konvertiert. Beim textuellen Schreiben wird der Umbruch auf den entsprechenden Systemen wieder zu den zwei Zeichen expandiert. Die binären Varianten filtern nichts und modifizieren nichts. Es wird alles Byte für Byte transportiert. In Listing 5.13 wird ein einfaches Programm gezeigt, das in der Lage ist, eine Datei zu kopieren. Mit dem Flag `ios::binary` müssen noch die Flags `ios::in` und `ios::out` angegeben werden, denn diese sind die Vorgabewerte für die zweiparametrigen Konstruktoren der File-Streamklassen.

Listing 5.13. Einfaches Kopierprogramm mit File-Streams

```
#include <fstream>

int main(int argc, char *argv[])
{
  using namespace std;

  if( argc != 3 ) return 0;

  ifstream in( argv[1], ios::binary | ios::in );
  ofstream out( argv[2], ios::binary | ios::out );

  if( in && out )
  {
    char c;
    while( in.get( c ) )
      out.put( c );
```

[3] Das sind vor allem die Windows-Systeme.

```
  }

  return 0;
}
```

Das Programm aus Listing 5.13 kann mit der folgenden Kommandozeile aufgerufen werden: `prg infile outfile`. Es ist ein einfaches Kopierprogramm, das eine Datei in eine andere kopiert. Die Methoden `get()` und `put()` verarbeiten jedes Byte und übergehen keine Whitespaces[4].

5.2.5 Die String- Streams

Die String-Streams arbeiten auf Textstrings. Dabei werden drei verschiedene Streamtypen unterschieden: der Eingabe-, der Ausgabe- und der IO-Stream (Ein- und Ausgabe).

```
istringstream
ostringstream
stringstream
```

Anhand des Ausgabestreams soll gezeigt werden, dass sich die Anwendung eines solchen Streams nicht von anderen Streamarten unterscheidet. Einen `ostringstream` kann man zum Beispiel als Textpuffer verwenden.

Listing 5.14. Anwendung eines `ostringstream`-Objekts als Textpuffer

```
#include <iostream>
#include <sstream>
#include <iomanip>

int main()
{
  using namespace std;
  ostringstream str;

  str << right;

  for( int i = 1; i < 7; ++i )
  {
    str << setw(7) << (i*i*i) << endl;
  }
```

[4] Unter *Whitespaces* versteht man Leerzeichen, Tabulatoren und Zeilenumbrüche. Diese Zeichen trennen bei einem `istream`-Operator » die ausgewerteten Bereiche. Insofern werden die Whitespaces selbst von den Einlesoperatoren nicht erfasst.

```
  cout << str.str();

  return 0;
}
```

Die Ausgabe des Beispiels in Listing 5.14 gleicht der des Beispiels in Listing 5.10 mit dem einen Unterschied, dass nicht direkt auf die Standardausgabe geschrieben wird, sondern zuerst in den Puffer, den der String-Stream zur Verfügung stellt, und danach erst auf die Standardausgabe.

Um die String-Streams nutzen zu können, muss man die Headerdatei `<sstream>` inkludieren.

Das nächste Beispiel soll zeigen, dass die Klasse `istringstream` dazu verwendet wird, Texte mit Hilfe von Eingabeoperatoren zu zerlegen. Im Beispiel in Listing 5.15 werden aus einem einfachen Text zwei Zahlen extrahiert.

Listing 5.15. Anwendung des `istringstream` zur Eingabe aus Strings

```
#include <iostream>
#include <sstream>

struct Complex
{
  double r;
  double i;
};

std::istream& operator>>( std::istream &is, Complex &c )
{
  is >> c.r;
  is >> c.i;
  return is;
}

std::ostream& operator<<( std::ostream &os,
                  const Complex &c )
{
  return os << "(" << c.r << "/i" << c.i << ")";
}

int main()
{
  using namespace std;
  string txt = "5 3";
  istringstream str(txt);
```

```
  Complex c;

  str >> c;

  cout << c << endl;

  return 0;
}
```

Die String-Streams eignen sich hervorragend dafür, Texte zu montieren oder zu zerlegen. Verwendet man bei einem Eingabestream die Streamoperatoren, gewinnt man damit die automatische Strukturierung anhand der Whitespaces. Möchte man diese Strukturierung nicht, kann man auf Methoden der Eingabestreams zurückgreifen.

5.2.6 Die STL

STL steht für Standard Template Library und enthält auf Templates basierende Implementierungen von Datencontainern, dazugehörigen Zugriffsmechanismen und Algorithmen. Die STL ist seit 1998 ein Teil der C++-Standardbibliothek, obwohl sie ursprünglich außerhalb der Standardlinie der Bibliothek entwickelt wurde. Sie entstand Anfang der neunziger Jahre. Damals lieferten die verschiedenen Hersteller von Entwicklungssystemen eigene Containerbibliotheksimplementierungen mit ihren spezifischen C++-Bibliotheken aus. Das Problem war dabei die fehlende Portierbarkeit, denn jeder Hersteller hatte seine ganz eigene Implementierung der gleichen allgemeinen Datenstrukturen. Dabei wurden auch häufig Compilerspezifika verwendet, die per se nicht auf anderen Compilern zur Verfügung standen. Es gab außerdem noch eine Diskussion darüber, ob Containerbibliotheken besser mit objektorientierten Technologien zu implementieren seien, oder vielleicht mit den neuen generischen Methoden. Es gab Vertreter beider Richtungen, bis Stefanov[5] die Diskussion im Wesentlichen zu Gunsten der generischen Methoden entschied.

Containertypen

Die STL definiert fünf wesentliche Grunddatenstrukturen zur Realisierung von 1-zu-n-Beziehungen im OO-Design: `list`, `vector`, `deque`, `set` und `map`. Der Containertyp `list` stellt eine doppelt verkettete Liste dar. Durch `vector` wird ein sich durch Reallokation dynamisch erweiternder Vektor definiert. Ähnlich der Liste ist `deque` ein Sequenzcontainer, der ein schnelles Einfü-

[5] Alexander Stefanov, der Autor der STL. Er implementierte die Bibliothek Anfang der neunziger Jahre mit Unterstützung von HP.

gen an beiden Enden ermöglicht. Allerdings ist auch der indizierte Zugriff in konstanter Zeit möglich. Der Container `set` ist als Binärer Baum realisiert. Er sortiert seine Elemente beim Einfügen und erlaubt auch nicht mehrere identische Werte. Der letzte im Bunde ist ein assoziativer Container. Die `map` speichert Datenpaare mit Schlüssel- und Wertedaten. Über die Schlüsseldaten werden die Elemente sortiert. Auch die Map erlaubt nur eineindeutige Daten. Sowohl der Container `set` als auch `map` haben jeweils ein Pendant, das nicht eineindeutige Elemente verlangt, sondern auch mehrere Elemente des gleichen Wertes aufnehmen kann. Für `set` ist das der Container `multiset` und für `map` entsprechend `multimap`.

Aufbauend auf Basiscontainern definiert die STL so genannte Adapterklassen, die Container für bestimmte Anwendungsbereiche darstellen, wie z. B. eine Queue – `queue` –, eine Prioritäts-Queue – `priority_queue` – oder einen Stack – `stack`.

Die Basiscontainertypen können mit so genannten Iteratoren zugegriffen werden. Container-Adapter haben keine Iteratoren.

Iteratoren sind Datentypen, die einen „Zugriff" definieren, wie es im Abschnitt 3.2.1 beschrieben ist. Sie verhalten sich in der STL aufgrund von überladenen Operatoren wie Zeiger. Auch einen Zeiger kann man als „Zugriff" auffassen. Bei bestimmten Containern ist der zugehörige Iterator tatsächlich als Zeiger definiert. Neben den Container- und Iteratorklassen gibt es noch die Algorithmen, die auf Container angewendet werden können. Diese sind aber durch die Iteratorschicht vollkommen von den Containern getrennt. Das heißt, dass ein Algorithmus auf einen beliebigen Container angewendet werden kann, ohne dessen interne Struktur und Datenhaltung kennen zu müssen. Wie das funktioniert, soll in den nachfolgenden Abschnitten verdeutlicht werden.

Die einfache Anwendung von Iteratoren auf Datenstrukturen

Je nach Art des Zugriffs kennen die Container unterschiedliche Iteratoren. So gibt es beispielsweise einen `const_iterator`, der nur für den lesenden Zugriff verwendbar ist. Wenn ein Container eine `const`-Maskierung aufweist, darf sein Inhalt nicht verändert werden. Das wird dadurch erreicht, dass das konstante Containerobjekt mit der Methode `begin()` einen `const_iterator` zurückliefert, über den nur gelesen werden kann. Um dies zu erreichen wendeten die Entwickler der STL einen Trick an, der darin besteht, dass an konstanten Objekten nur Methoden aufgerufen werden können, die den `this`-Zeiger konstant definieren. Die `const`-Spezifikation an Methoden zählt zu ihrer Signatur dazu. Das heißt, dass sie für ein Überladen von Methoden herangezogen werden kann. Die überladene Methode `begin()` mit der `const`-Spezifikation gibt einen anderen Iterator zurück als die ohne (den `const_iterator`).

```
iterator       begin ();
const_iterator begin () const;
```

```
iterator       end ();
const_iterator end ()   const;
```

Neben den Iteratoren `iterator` und `const_iterator` gibt es noch den `reverse_iterator` und den `const_reverse_iterator`. Diese werden von den Methoden `rbegin()` und `rend()` zurückgeliefert und dienen dazu, den Container in umgekehrter Reihenfolge zu durchlaufen. Auch diese Iteratoren besitzen einen überladenen Operator ++, der allerdings den Container in Gegenrichtung durchläuft.

```
reverse_iterator       rbegin ();
const_reverse_iterator rbegin () const;
reverse_iterator       rend ();
const_reverse_iterator rend ()   const;
```

Die entsprechenden Methoden, angewandt, sehen folgendermaßen aus:

Listing 5.16. Anwendung von Iteratoren in der STL

```
#include <iostream>
#include <list>

typedef std::list<int> Container;

void print_out( const Container &c )
{
    for( Container::const_iterator it = c.begin();
         it != c.end(); ++it )
        std::cout << *it;
    std::cout << std::endl;
}

void print_reverse( const Container &c )
{
    for( Container::const_reverse_iterator it
                                = c.rbegin();
         it != c.rend(); ++it )
        std::cout << *it;
    std::cout << std::endl;
}

int main()
{
    Container l;

    l.push_back( 3 );
```

```
    l.push_back( 5 );
    l.push_back( 7 );

    print_out(l);
    print_reverse(l);

    return 0;
}
```

Die beiden rückläufigen Iteratoren drehen die Datenhaltung des Containers natürlich nicht physikalisch um. Sie stellen lediglich eine Abstraktion dar, die den Container in seiner Sequenz anders herum betrachtet.

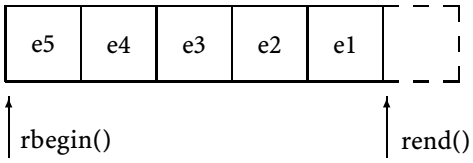


Abbildung 5.1. Logische Elementsequenz bei der Verwendung von Reverse-Iteratoren

Die beschriebenen Iteratoren haben sowohl einen überladenen Inkrement- als auch einen Dekrementoperator. Man kann sie also in zwei Richtungen bewegen. Aus diesem Grund spricht man von ihnen auch als bidirektionalen Iteratoren[6]. Eine flexiblere Variante ist der Random-Access-Iterator, den man um beliebige Schrittweiten verschieben kann. Ein solcher Iterator akzeptiert beispielsweise eine Addition mit `5`, was einem Verschieben um fünf Positionen entspricht. Der Standardvektor aus der STL hat solche Random-Access-Iteratoren. Grundsätzlich kann man solche Iteratoren auch für andere Container entwickeln. Aus Laufzeitgründen hat die Liste keinen Random-Access-Iterator. Welche Iteratoren es in der STL gibt und in welchem Zusammenhang sie zueinander stehen, wird im Abschnitt 5.2.6 genauer behandelt.

`std::list`

Ein wesentlicher Container, den die STL zur Verfügung stellt, ist die Liste. Die Liste wird durch das Klassentemplate `list` repräsentiert. Das erste Templateargument dieser Klasse bestimmt den Inhaltstyp der Liste. Der zweite Templateparameter beschreibt die so genannte Allokatorklasse und muss nicht angegeben werden, da er einen Vorgabetyp besitzt. Die Allokatorklasse definiert die Strategie, mit der Speicher für die Listenelemente allokiert wird.

Im folgenden Beispiel wird eine Liste instanziiert, mit Daten gefüllt und wieder ausgelesen.

[6] Es gibt grundsätzlich auch reine Forward-Iteratoren.

Listing 5.17. Kurze Demonstration der STL-Liste

```
#include <iostream>
#include <list>

int main()
{
  std::list<int> l;

  l.push_back( 3 );
  l.push_back( 5 );
  l.push_back( 7 );

  for( std::list<int>::iterator it = l.begin();
       it != l.end();
       ++it )
       std::cout << *it << std::endl;

  return 0;
}
```

Eine Besonderheit bei der STL ist, dass man die Iteratortypen immer als eingeschachtelte Typen des Containers bekommt. Das ist eine enorme Erleichterung in der Arbeit mit Iteratoren. Wären sie nicht im Containertyp greifbar, müsste man bei jeder Verwendung den richtigen Iteratortyp dem richtigen Container zuordnen. Bei der STL entscheidet man sich für einen Container und bekommt die Iteratortypen mitgeliefert. Ein einfacher bidirektionaler Iterator, ein Iterator also, mit dem man in beide Richtungen laufen kann, heißt in einem STL-Container beispielsweise `iterator`. Eine weitere Besonderheit ist, dass man, um das Ende einer Collection abzufragen, auf den Ende-Iterator überprüft. Der Iterator, der von der Methode `end()` zurückgeliefert wird, verweist auf ein Element hinter dem letzten Listenelement. Das heißt, dass der Inhalt, auf den dieser Ende-Iterator verweist, nicht gültig ist. Man darf einen solchen Iterator niemals mit dem Sternoperator dereferenzieren. Diese Zugriffstechnik auf die Container arbeitet also mit einem offenen Intervall.

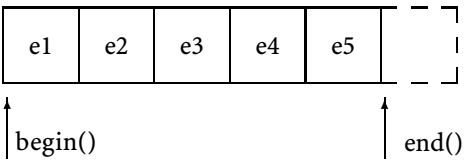


Abbildung 5.2. Das halb offene Intervall bei der Verwendung von `begin()` und `end()`

Die Zeichnung verdeutlicht das Prinzip, das in der STL durchgängig angewandt wird. In folgenden Abschnitten wird noch verdeutlicht werden, welcher Vorteil in der stringenten Anpassung dieses Prinzips auf alle Containertypen liegt.

std::vector

Ein weiterer Grundcontainer der STL ist der Vektor, der in der Klasse `vector` implementiert ist. Auch er hat zwei Templateargumente, wobei das zweite, den Allokator betreffende, vorbelegt ist.

Das Codebeispiel aus dem Beispiel it der Liste kann mit einfachen Mitteln auf den Vektor angepasst werden. Grundsätzlich funktioniert das Füllen und das Zugreifen auf den Vektor genauso wie auf die Liste, obwohl eine ganz andere Datenhaltung zugrunde liegt.

Listing 5.18. Eine Demonstration des STL-Vektors

```
#include <iostream>
#include <vector>

int main()
{
  std::vector<int> l;

  l.push_back( 3 );
  l.push_back( 5 );
  l.push_back( 7 );

  for( std::vector<int>::iterator it = l.begin();
       it != l.end();
       ++it )
       std::cout << *it << std::endl;

  return 0;
}
```

Die identischen Zugriffstechniken eröffnen weitere Freiheitsgrade im Entwurf von Software. Man kann zum Beispiel durch geschicktes Anbringen von `typedef`-Anweisungen den Clientcode ganz vom Typebezeichner des Containers freihalten:

Listing 5.19. Variierung des Datencontainers im fachlichen Code

```
#include <iostream>
#include <list>
#include <vector>
```

```
// typedef std::list<int> MyContainer;
typedef std::vector<int> MyContainer;

int main()
{
  MyContainer l;

  l.push_back( 3 );
  l.push_back( 5 );
  l.push_back( 7 );

  for( MyContainer::iterator it = l.begin();
       it != l.end();
       ++it )
       std::cout << *it << std::endl;

  return 0;
}
```

Damit lässt sich beispielsweise experimentieren. Oder man kann dem Clientcode eine ganz neue Containerimplementierung zur Verfügung stellen.

In ihrer Datenhaltung unterscheiden sich die Containertypen allerdings, was natürlich auch in unterschiedlichen Schnittstellen zum Ausdruck kommt. Neben den verallgemeinerten Zugriffsmechanismen über die Iteratoren haben die STL-Container Methoden, die ihrer internen Datenhaltung gerecht werden und für den speziellen Containertyp sehr effektive Operationen zur Verfügung stellen. Für den Typ `vector` ist das beispielsweise der indizierte Zugriff über die Methode `at()` oder den überladenen Indexoperator `[]`. Dabei unterscheidet sich die Methode `at()` von den Indexklammern `[]` dadurch, dass sie eine Ausnahme werfen kann, wenn der Indexbereich überschritten wird.

Listing 5.20. Drei mögliche Zugriffsarten auf den Vektor

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> l;

    l.push_back( 3 );
    l.push_back( 5 );
    l.push_back( 7 );
```

```
    for( std::vector<int>::iterator it = l.begin();
         it != l.end();
         ++it )
        std::cout << *it << std::endl;

    std::cout << std::endl;

    for( int i = 0; i < l.size(); i++ )
        std::cout << l[i] << std::endl;

    std::cout << std::endl;

    for( int j = 0; j < l.size(); j++ )
        std::cout << l.at(j) << std::endl;

    return 0;
}
```

Wie im Abschnitt 3.3 näher behandelt wurde, ist ein Vektor immer ein zusammenhängender Speicherbereich. Er besitzt eine Größe und eine Kapazität. Wenn für das Anfügen eines Elements eine Vergrößerung der Kapazität notwendig ist, wird ein neuer zusammenhängender Speicherbereich allokiert und der gesamte Inhalt des alten umkopiert. Der zusammenhängende Speicherbereich macht den Zugriff über einen Indexwert besonders effektiv[7].

std::set

Der dritte Grundtyp eines Containers in der STL ist eine sortierte Menge. Implementiert ist die sortierte Menge in dem Klassentemplate `set`.

Der Datentyp `set` ist von seiner internen Struktur ein Binärer Baum. Daher eignet er sich besonders gut für Mengen, die sortiert aufbewahrt werden müssen. Dafür wird an den Inhaltstyp die Anforderung gestellt, dass der Operator `<` definiert sein muss, der einen Größenvergleich erlaubt. Im folgenden Beispiel wird der `int`-Datentyp als Inhaltstyp verwendet, für den ein solcher Operator existiert.

Listing 5.21. Ein Demonstrationsbeispiel zum STL-Set

```
#include <iostream>
#include <set>

int main()
```

[7] Im Gegensatz zur Liste, die sich aufgrund ihrer verketteten Struktur besonders gut für eine sequenzielle Traversierung eignet.

```
{
    using namespace std;

    set<int> s;

    s.insert( 3 );
    s.insert( 4 );
    s.insert( 1 );
    s.insert( 2 );

    for( set<int>::iterator it = s.begin();
         it != s.end(); ++it )
        cout << *it << endl;

    return 0;
}
```

Der Container `set` ist aufgrund seiner Eigenschaft, dass er die Elemente beim Einfügen sortiert, eine ideale Grundlagenimplementierung für alle Datencontainer, die ihre Daten sortiert aufbewahren müssen.

Seine wesentlichen Merkmale sind die Struktur der Datenhaltung in Form eines Binären Baums und der Vergleich eines neuen einzufügenden Elements mit schon darin befindlichen Elementen.

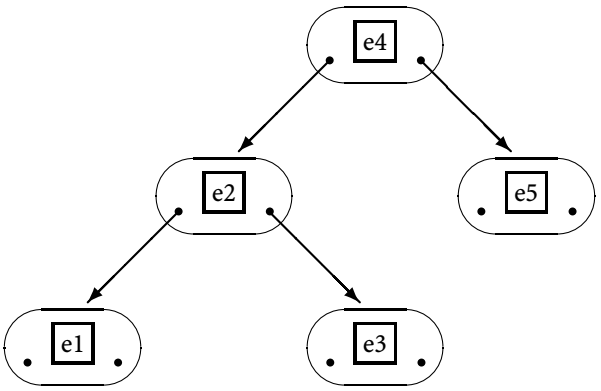


Abbildung 5.3. Organisationsprinzip des Binären Baums

Ein Element, das schon einmal in einem Set enthalten ist, wird kein zweites Mal aufgenommen.

`std::map`

Das erste einfache Beispiel zur Verwaltung von Schlüssel-Werte-Paaren soll ein Mapping zwischen englischen und deutschen Worten implementieren.

Listing 5.22. Ein Demobeispiel zur STL-Map

```
#include <iostream>
#include <map>
#include <string>

int main()
{
  using namespace std;
  map<string,string> dictionary;

  dictionary["monitor"]  = "Bildschirm";
  dictionary["keyboard"] = "Tastatur";
  dictionary["mouse"]    = "Maus";

  cout << dictionary["keyboard"] << endl;

  return 0;
}
```

Das Programm erzeugt die Ausgabe „`Tastatur`“.

Das nachfolgende Beispiel soll auf einfache Weise demonstrieren, wie eine Map noch angewandt werden kann. Es soll einfach die Häufigkeit der vorkommenden Wörter zählen und die Wörter in alphabetischer Reihenfolge mitsamt der gezählten Häufigkeit ausgeben.

Die Map benötigt mindestens zwei Templateparameter. Einen für den Schlüsseldatentyp und einen für den Wertetyp. Mit diesen beiden Typen wird innerhalb der Map ein Datentyp für das Datenpaar kreiert, mit dem die Map als eigentlichem Elementtyp arbeitet: `pair<key_type,value_type>`. In dem Fall des nachfolgenden Beispiels ist es der Typ `pair<string,unsigned int>`. Wenn man an einem solchen Datenpaar die Elemente lesen möchte, muss man auf die Attribute `first` und `second` zugreifen.

Listing 5.23. Ein Programm zum Zählen von Wörtern in Texten

```
#include <iostream>
#include <string>
#include <map>

int main()
{
  using namespace std;
```

```
  typedef map<string,unsigned int> WordMap;
  WordMap m;

  while( !cin.eof() )
  {
    string s;
    cin >> s;
    if( !cin.eof() )
      m[s] += 1; // Hinzufügen und aufzählen.
  }

  cout << "Wort\tHaeufigkeit" << endl;
  for( WordMap::iterator i = m.begin();
       i != m.end(); ++i )
  {
    cout << i->first << "\t" << i->second << endl;
  }

  return 0;
}
```

Das Besondere an der Map ist ihr typischer Indexoperator `[]`. Dieser Operator kann nicht nur ein vorhandenes Element anhand eines Schlüssels finden, er kann auch ein Element anlegen, wenn es noch nicht vorhanden ist. Dabei geht dieser Container nur nach dem Vorhandensein eines Schlüsselwertes. In unserem Beispiel sind die Schlüsselwerte Strings. Ist ein bestimmter String noch nicht in der Map enthalten, fügt der Operator ein neues Datenpaar ein. Der Inhaltswert ist zunächst 0, wenn nichts anderes angegeben wurde. Dieser Inhaltswert wird aber nun um eins erhöht. Wenn bei einem weiteren Durchlauf der Schleife ein weiteres Mal der gleiche String gefunden wurde, gibt der Operator den schon vorhandenen Inhaltswert zur weiteren Manipulation zurück.

Container- Adapter

Die STL stellt einige abstrakte Datentypen zur Verfügung, die Container auf einen bestimmten Verwendungszweck hin adaptieren. Diese Adapter verbergen den zugrunde liegenden Container und stellen ganz bestimmte Funktionen bereit, die der durch Adaption gewonnenen Funktionsweise entsprechen. Sie verbergen also die native Schnittstelle des Containers und stellen dafür eine andere bereit. Solche Adaptertypen sind `stack`, `queue` und `priority_queue`.

`std::stack`

Ein Stack[8] implementiert eine LIFO-Funktionalität[9]. Man kann sich die Verwendung eines Stacks wie die eines typischen Stapels vorstellen: Man legt Elemente oben auf, um sie dann als erste wieder herunterzunehmen. Die weiter unten liegenden Elemente wurden vor den oberen auf den Stapel gelegt und werden auch später wieder heruntergenommen. Es gibt eine ganze Reihe von Situationen in der Softwareentwicklung, die die Verwendung eines Stacks nötig machen. Die wesentlichen, dafür bereitgestellten Methoden sind `push()`, `pop()` und `top()`. Da immer nur an einem Ende hinzugefügt und herausgenommen wird, ist der Container `deque` dafür bestens geeignet. Das ist der Grund dafür, dass der Stack standardmäßig mit `deque` als Vorgabecontainer arbeitet. Die Deklaration des Stacks sieht deshalb folgendermaßen aus:

```
template <class T, class Container = deque<T> >
class stack;
```

Der Vorgabecontainer `deque` kann jedoch jederzeit durch einen anderen ersetzt werden. Voraussetzung dafür ist jedoch, dass der Container die Methoden `push_back()`, `pop_back()` und `back()` besitzt. Diese Voraussetzung ist auch bei der Liste und dem Vektor in der STL gegeben. Deshalb lassen sich Stacks auch mit diesen zwei Containertypen instanziieren.

Listing 5.24. Ein Demobeispiel zum STL-Stack

```
#include <iostream>
#include <string>
#include <list>
#include <vector>
#include <deque>
#include <stack>

template <typename STACK>
void WriteOut( STACK &stack )
{
  using std::cout;
  using std::endl;
  while( stack.size() )
  {
    cout << stack.top() << endl;
    stack.pop();
  }
}
```

[8] stack engl. = Stapel.

[9] **L**ast **I**n **F**irst **O**ut. Die Elemente, die zuletzt eingefügt wurden, werden zuerst herausgenommen.

```
int main()
{
  using namespace std;
  stack< string, vector<string> > vs;
  stack< string, list<string> >   ls;
  stack< string, deque<string> >  ds;

  vs.push( "Erdbeere" );
  vs.push( "Aprikose" );
  vs.push( "Kirsche" );

  ls.push( "Kartoffel" );
  ls.push( "Kohlrabi" );
  ls.push( "Zwiebel" );

  ds.push( "Bohne" );
  ds.push( "Linse" );
  ds.push( "Erbse" );

  WriteOut( vs );
  cout << "***" << endl;
  WriteOut( ls );
  cout << "***" << endl;
  WriteOut( ds );

  return 0;
}
```

Das Beispielprogramm erzeugt die Ausgabe:

```
Kirsche
Aprikose
Erdbeere
***
Zwiebel
Kohlrabi
Kartoffel
***
Erbse
Linse
Bohne
```

Man sieht also, dass der Stack auf den unterschiedlichen Basis-Containern funktional nach außen identisch ist. Er unterscheidet sich in seiner inneren Implementierung, was Konsequenzen für den Systemkontext hat, in dem er

läuft. Je nach Basis-Container verhält sich der Stack anders bezüglich der Speicherallokation und der Laufzeit. Eine gleichbleibende Laufzeit wird er bei der Verwendung einer Liste aufweisen, da die Liste selbst für Einfüge- und Löschoperationen eine von der Elementanzahl unabhängige Laufzeit aufweist. Anders reagiert der Stack, der auf einem Vektor basiert. Der Vektor hat eine von der Elementanzahl abhängige Laufzeit bei Operationen, die seine Größe verändern. Dafür allokiert der Vektor den benötigten Speicher in einem Block, während die Liste den Speicher für jedes Element separat allokiert. Diese Überlegungen sind es, die man sich für die Auswahl des richtigen Stacktyps machen muss. Vor allem dann, wenn die beiden Faktoren Laufzeit und Speicher für ein gestelltes Problem stark limitierend sind.

`std::queue`

Der Container `queue` stellt eine Funktionalität zur Verfügung, die für alle möglichen Implementierungen von Queues benötigt werden. Queues – zu deutsch „Warteschlangen" – arbeiten nach dem FIFO-Prinzip[10]. Die Elemente, die zuerst in die Warteschlange eingestellt werden, werden auch zuerst wieder herausgenommen. Dafür gibt es die Methoden `push()` und `pop()` sowie `front()` und `back()`. Während `push()` wie auch beim Stack hinten an der Queue ein neues Element anfügt, reagiert `pop()` anders als die gleichnamige Methode im Stack. Bei der Queue löscht sie das erste Element, beim Stack das letzte. Da es besonders auf die Einfügeoperationen ankommt, ist die Queue standardmäßig als `deque` implementiert.

```
template <class T, class Container = deque<T> > class queue;
```

Es kann aber auch die Liste aus der STL für eine Queue verwendet werden, wie im folgenden Beispiel gezeigt wird:

Listing 5.25. Ein Demobeispiel zur STL-Queue

```
#include <iostream>
#include <string>
#include <list>
#include <deque>
#include <queue>

template <typename Q>
void WriteOut( Q &q )
{
  using std::cout;
  using std::endl;
  while( q.size() )
  {
```

[10] First In First Out.

```
    cout << q.front() << endl;
    q.pop();
  }
}

int main()
{
  using namespace std;
  queue< string, list<string> >  lq;
  queue< string, deque<string> > dq;

  lq.push( "Kartoffel" );
  lq.push( "Kohlrabi" );
  lq.push( "Zwiebel" );

  dq.push( "Bohne" );
  dq.push( "Linse" );
  dq.push( "Erbse" );

  WriteOut( lq );
  cout << "***" << endl;
  WriteOut( dq );

  return 0;
}
```

Das Programm erzeugt die Ausgabe:

```
Kartoffel
Kohlrabi
Zwiebel
***
Bohne
Linse
Erbse
```

std::priority_queue

Eine `priority_queue` verhält sich ganz ähnlich wie eine `queue`, mit dem wesentlichen Unterschied, dass die Elemente sortiert eingeordnet werden. Die Elemente in der Prioritätswarteschlange werden also nicht nach dem FIFO-Prinzip behandelt, sondern nach der Reihenfolge von Prioritäten. Eine Priorisierung wird nach bestimmten fachlichen Gesichtspunkten durchgeführt. Was für eine Prioritätswarteschlange aber immer gleich bleibt, ist, dass eine Auswertung der Prioritäten durchgeführt werden muss und dass diese Aus-

wertung zu einer Sortierung der Elemente innerhalb der Warteschlange führen muss. Genau diesen Vorgang implementiert `priority_queue` in der STL.

Um die Sortierung durchzuführen, muss dem Containeradapter ein Typ mitgeliefert werden, der einen Vergleich zwischen zwei Elementen ermöglicht. Mit diesem Vergleich bestimmt man die Art der Sortierung. Standardmäßig kommt dabei der Vergleichstyp `less` als Vorgabe zum Einsatz. Vorgabecontainer ist der Vektor aus der STL. Neben `vector` nimmt der Adapter `priority_queue` auch `deque` als Implementierungsgrundlage.

```
template <class T, class Container = vector<T>,
class Compare = less<typename Container::value_type> >
class priority_queue;
```

Listing 5.26. Ein Demobeispiel zur STL-Priority Queue

```
#include <iostream>
#include <string>
#include <vector>
#include <deque>
#include <queue>

template <typename Q>
void WriteOut( Q &q )
{
  using std::cout;
  using std::endl;
  while( q.size() )
  {
    cout << q.top() << endl;
    q.pop();
  }
}

int main()
{
  using namespace std;
  priority_queue< string, vector<string> > vq;
  priority_queue< string, deque<string> >  dq;

  vq.push( "Erdbeere" );
  vq.push( "Apprikose" );
  vq.push( "Kirsche" );

  dq.push( "Bohne" );
  dq.push( "Linse" );
```

```
  dq.push( "Erbse" );

  WriteOut( vq );
  cout << "***" << endl;
  WriteOut( dq );

  return 0;
}
```

Die Ausgabe des Programms:

```
Kirsche
Erdbeere
Apprikose
***
Linse
Erbse
Bohne
```

Werden die Templates mit einem anderen Vergleichstyp parametrisiert,

```
priority_queue< string, vector<string>, greater<string> > vq;
priority_queue< string, deque<string>, greater<string> >  dq;
```

verändert sich auch das Ergebnis.

```
Apprikose
Erdbeere
Kirsche
***
Bohne
Erbse
Linse
```

Die Vergleichstypen `less` und `greater` basieren darauf, dass die zu vergleichenden Elementtypen den Operator „<“ besitzen. Dieser Operator wird innerhalb der Funktoren `less` und `greater` aufgerufen. Sollen Elemente verglichen werden, die diesen Operator nicht besitzen, so muss ein eigener Vergleichstyp geschrieben werden, der in einer eigenen Weise die Elemente miteinander vergleicht.

Die Iteratoren

Die Container der STL werden soweit es geht mit Iteratoren traversiert. Andere Zugriffsmöglichkeiten auf einen Datencontainer eröffnen sich immer dann, wenn die Art des Zugriffs eng mit der inneren Datenhaltung zusammenhängt. Das Grundkonzept des Iterators als „Zugriffsklasse“ wurde in Abschnitt 3.2.1

auf Seite 178 erklärt. Etwas von der praktischen Anwendung der Iteratoren wurde schon in den vorangegangenen Abschnitten gezeigt. Hier soll nun das Iteratorkonzept in der STL etwas genauer betrachtet werden.

Je nach Datencontainer und Anwendung werden unterschiedliche Iteratoren gebraucht. In der STL existieren daher fünf verschiedene Iteratorarten:

1. Input-Iteratoren
2. Output-Iteratoren
3. Forward-Iteratoren
4. Bidirectional-Iteratoren
5. Random-Access-Iteratoren

Input - Iterator

Input-Iteratoren erlauben einen lesenden Zugriff auf die Daten des traversierten Containers. Der Container kann dabei nur in einer Richtung durchlaufen werden. Ein Input-Iterator kann aber mit einem anderen Input-Iterator auf Gleichheit oder Ungleichheit überprüft werden. Er stellt die folgenden Operatoren zur Verfügung: ++, ==, !=, -> und * (nur lesend).

Output- - Iteratoren

Der Output-Iterator ist dazu da, Daten in Container oder Streams zu schreiben. Man kann mit ihm keine Elemente lesen. Er ist auch nicht auf Gleichheit oder Ungleichheit überprüfbar und definiert deshalb nur die Operatoren ++ und den Dereferenzierungsoperator * (nur schreibend).

Forward - Iteratoren

Ein Forward-Iterator kann wie ein Input-Iterator nur in eine Richtung – vorwärts – laufen. Dafür kann er aber lesend und schreibend auf die Elemente des Datencontainers zugreifen. Er definiert die Operatoren ++, ==, !=, -> und * (mehrfach lesend und schreibend).

Bidirectional - Iteratoren

Wie der Forward-Iterator unterstützt der Bidirectional-Iterator lesenden und schreibenden Zugriff auf die Datenelemente eines Containers. Zusätzlich erlaubt er das Traversieren in beiden Richtungen. Neben den Operatoren, die im Forward-Iterator schon verfügbar sind, definiert er noch den Dekrementoperator --.

Random - Access - Iteratoren

Der Random-Access-Iterator kann wie der Bidirectional-Iterator vorwärts und rückwärts laufen. Dazu kann man noch wie bei Zeigern eine Distanz zwischen zwei Iteratorpositionen bestimmen. Einen Distanzwert kann man

auf die aktuelle Iteratorposition addieren oder subtrahieren und damit die Position verändern. Dieser Iterator-Typ verhält sich damit wie Zeiger, die man auch mit Ganzzahlen verrechnen kann um neue Positionen zu erhalten. Der Random-Access-Iterator kennt die Operatoren ++, --, ==, !=, ->, *, +, -, +=, -=, <, >, <= , >= und [] . Der Klammeroperator [] ermöglicht einen indizierten Zugriff auf Elemente einer durch den Random-Access-Iterator traversierten Container.

Um die verschiedenen Iteratorkategorien zu kennzeichnen gibt es in der STL Markerinterfaces[11]. Die „Iterator-Tags“, wie man diese Markerinterfaces in der STL nennt, sind folgendermaßen definiert:

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag
       : public input_iterator_tag {};
struct bidirectional_iterator_tag
       : public forward_iterator_tag {};
struct random_access_iterator_tag
       : public bidirectional_iterator_tag {};
```

Die unterschiedlichen Iteratorkategorien haben also einen Zusammenhang, den man in einem Klassendiagramm darstellen kann:

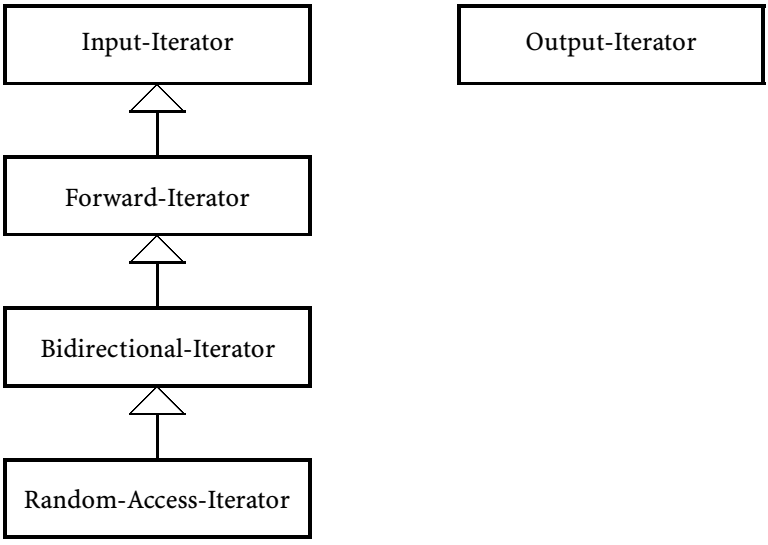


Abbildung 5.4. Schematische Darstellung der Beziehungen zwischen den Iteratortypen

[11] Markerinterfaces sind Basisklassen, die keine Methoden bereithalten und nur die Aufgabe haben, einen Typ dadurch zu kennzeichnen, dass er von einem solchen Markerinterface abgeleitet ist.

Die Mehrheit der Basiscontainer der STL: `list`, `map`, `multimap`, `set` und `multiset` haben Bidirectional-Iteratoren. Zwei der Basiscontainer: `deque` und `vector` haben Random-Access-Iteratoren.

Const - Iterator

Die verschiedenen Container definieren unterschiedliche Iteratortypen, je nach der Art der Datenhaltung. Dabei muss neben dem normalen Iterator auch noch ein konstanter Iterator durch den Container zur Verfügung gestellt werden, damit der Elementzugriff schreibgeschützt stattfinden kann, wenn der Container `const`-maskiert ist. Für alle Iteratorvarianten außer dem Output-Iterator gibt es die konstanten Pendants.

Reverse - Iterator

Eine spezielle Untergruppe von bidirektionalen und Random-Access-Iteratoren sind die Reverse-Iteratoren. Sie definieren einfach die Reihenfolge anders herum. In der STL gibt es also zu jedem bidirektionalen Iterator einen entsprechenden Reverse-Iterator, der die Richtungen vorwärts und rückwärts entgegengesetzt festlegt. Wenn man also eine Sequenz umdrehen möchte, braucht man das nicht physikalisch zu tun. Man greift einfach mit einem Reverse-Iterator auf die entsprechende Sequenz zu. Jeder Container, der also mindestens einen bidirektionalen Iterator zur Verfügung stellt, stellt auch einen entsprechenden Reverse-Iterator. Dafür gibt es dann extra Methoden, die Reverse-Iteratoren liefern – `rbegin()` und `rend()`.

Die Einfüge - Iteratoren

Eine wesentliche Anwendung von Output-Iteratoren in der STL sind die Einfüge-Iteratoren, mit denen man Elemente in Container einfügen kann. Die drei Iteratortypen dieser Art in der STL sind:

```
// C steht für "Container"
template <typename C> front_insert_iterator<C>;
template <typename C> back_insert_iterator<C>;
template <typename C> insert_iterator<C>;
```

Während der Konstruktor von `front_insert_iterator<>` ebenso wie der von `back_insert_iterator<>` nur den Container entgegennimmt, braucht der Konstruktor von `insert_iterator<>` noch einen Iterator, der die Position anzeigt, vor der neue Elemente eingefügt werden sollen. Man kann die Konstruktoren dieser Iteratoren nutzen oder die Funktions-Templates, die es ersparen, den Containertyp in spitzen Klammern beim Typ des Insert-Iterators anzugeben. Diese Funktions-Templates kapseln einfach den Konstruktoraufruf und detektieren den Typ des übergebenen Containers:

```
template <typename C>
front_insert_iterator<C> front_inserter(C &c);

template <typename C>
back_insert_iterator<C> back_inserter(C &c);

template <typename C, typename I>
insert_iterator<C> inserter(C &c, I pos );
```

In Listing 5.43 auf Seite 323 und in Listing 5.46 auf Seite 325 wird die praktische Anwendung von Einfüge-Iteratoren vorgestellt.

Stream - Iteratoren

Eine spezielle Sorte von Iteratoren sorgt für die Anbindung der Streams der Standardbibliothek an die STL. Die Stream-Iteratoren betrachten Streams wie Datencontainer. Mit dem Beispielprogramm in Listing 5.27 wird mit einem `ostream_iterator<>` in eine Datei geschrieben. Der Iterator definiert den Dereferenzierungsoperator `*` und den Inkrementoperator `++` wie alle anderen Iteratoren der STL auch.

Listing 5.27. Erstes Beispiel zu einem Stream-Iterator

```
#include <fstream>
#include <iterator>
#include <string>

int main()
{
  using namespace std;

  ofstream outd( "test.txt" );
  ostream_iterator<string> outs( outd );

  *outs = "Die erste Zeile Text\n";
  ++outs;
  *outs = "Die zweite Zeile Text\n";
  ++outs;
  *outs = "Die dritte Zeile Text\n";

  return 0;
}
```

In der Textdatei muss nach der Ausführung des Programms der folgende Text stehen:

```
Die erste Zeile Text
Die zweite Zeile Text
Die dritte Zeile Text
```

Der `ostream_iterator<>` kann auf jeden beliebigen `ostream` angewendet werden; also auch auf einen `ostringstream` oder einfach auf `cout`.

Im nächsten Beispiel soll die Funktionsweise des `istream_iterator<>` gezeigt werden. Das Programm in Listing 5.28 zählt die Wörter in einem übergebenen Text.

Listing 5.28. Ein weiterer Wörterzähler mit Streamiteratoren

```
#include <iostream>
#include <fstream>
#include <iterator>
#include <string>
#include <map>

int main()
{
  using namespace std;
  map<string,unsigned> wordmap;
  istream_iterator<string> ins( cin ), endofstream;

  while( ins != endofstream )
  {
    string word = *ins;
    wordmap[ word ] += 1;
    ++ins;
  }

  for( map<string,unsigned>::
       iterator it=wordmap.begin();
       it != wordmap.end(); ++it )
  {
    cout << it->first
         << "\t"
         << it->second
         << endl;
  }

  return 0;
}
```

Auch der `istream_iterator<>` hat die Operatoren `*` und `++` und verhält sich damit wie jeder andere Iterator aus der STL. Mit diesen Operatoren tra-

versiert man den Iterator und greift auf den Inhalt der zugrunde liegenden Datenstruktur zu. Mit dem Aufruf des Programms kann man eine Datei in die Standardeingabe lenken. Wenn man dazu die Ausgabedatei aus dem vorangegangenen Beispiel nimmt, `prg < test.txt`, erzeugt das Programm die Ausgabe:

```
Die     3
Text    3
Zeile   3
dritte  1
erste   1
zweite  1
```

Das Programm baut auf den Fähigkeiten des Datencontainers `map` auf, wie er im Abschnitt 5.2.6 ab Seite 301 beschrieben ist.

Das nächste Beispiel soll zeigen, dass man mit der Art der Iteratoren die Datentypen festlegen kann, die man aus einer Textdatei lesen möchte. Mit dem Iterator legt man damit auch die Methode der Interpretation des Textes fest. Im Beispiel in Listing 5.29 wird ein `istream_iterator<>` mit dem Typ `int` parametrisiert. Damit erwartet er Zahlen in dem einzulesenden Stream. Die eingelesenen Daten werden gleich wieder über einen `ostream_iterator<>` auf der Standardausgabe ausgegeben. Allerdings wurde dieser Iterator mit einem zweiten Parameter im Konstruktor initialisiert, der eine Trennzeichensequenz nach den einzelnen Elementen ausgibt.

Listing 5.29. Umformatierung von Streams über Iteratoren

```
#include <iostream>
#include <fstream>
#include <iterator>

int main()
{
  using namespace std;
  ifstream ind( "dat1.txt" );
  istream_iterator<int> ins( ind ), endofstream;
  ostream_iterator<int> outs( cout, " - " );

  while( ins != endofstream )
  {
    *outs++ = *ins++;
  }

  return 0;
}
```

Wenn die Datei `dat1.txt` folgende Zahlen enthält:

```
12
42
3
99
37
```

dann hat das Programm die Ausgabe: `12 - 42 - 3 - 99 - 37 - `. Man kann zum Beispiel durch die Angabe von `"\n"` als zweitem Konstruktorparameter Zeilenumbrüche erzwingen lassen.

Interessant wird der Einsatz von Stream-Iteratoren bei der Anwendung der Algorithmen, die im nachfolgenden Abschnitt beschrieben werden. Es können durch die Iteratorschnittstelle die Operationen, die auf Datencontainern möglich sind, auch auf Streams angewandt werden.

Algorithmen

Die Iteratorschnittstelle der Container macht es möglich, dass von den Datenhaltungen unabhängige Algorithmen definiert werden, die sequenziell arbeiten. Die STL definiert solche Algorithmen in Form von Funktionstemplates, die eine Datensequenz in Form von Iteratoren erwarten und oft auch eine Operation in Form eines Funktorobjekts übergeben bekommen. Eine solche Operation kann auch Teil eines Algorithmus sein. Einen solchen Teilalgorithmus nennt man in der STL Prädikat.

Listing 5.30. Der Algorithmus `for_each`

```
#include <iostream>
#include <set>
#include <algorithm>

void ausgabe( int i )
{
    std::cout << i << std::endl;
}

int main()
{
    using namespace std;

    set<int> s;

    s.insert( 3 );
    s.insert( 4 );
    s.insert( 1 );
```

```
    s.insert( 2 );

    for_each( s.begin(), s.end(), ausgabe );

    return 0;
}
```

In dem Beispiel aus Listing 5.30 wurde der Funktionszeiger `ausgabe` als Prädikat für den Algorithmus `for_each<>()` verwendet. Ein solches Prädikat kann auch ein Objekt sein. Wichtig ist nur, dass es sich wie ein Funktionszeiger verhält. Dazu kann der Klammeroperator überladen werden. Ein Objekt, das sich wie eine Funktion verhält und sich deshalb als Prädikat verwenden lässt, nennt man auch Funktorobjekt. Zur Übergabe des Prädikats an den Algorithmus muss das Prädikatobjekt instanziiert werden. Im folgenden Beispiel in Listing 5.31 wird dazu der Standardkonstruktor aufgerufen.

Listing 5.31. Das Prädikat als Klassenobjekt

```
#include <iostream>
#include <set>
#include <algorithm>

class ausgabe
{
public:
    void operator()( int i )
    {
        std::cout << i << std::endl;
    }
};

int main()
{
    using namespace std;

    set<int> s;

    s.insert( 3 );
    s.insert( 4 );
    s.insert( 1 );
    s.insert( 2 );

    for_each( s.begin(), s.end(), ausgabe() );

    return 0;
}
```

Der Vorteil eines Prädikats, das als Klasse implementiert wird, besteht darin, dass ein Objekt dieser Klasse Daten mitführen kann. Die Funktion, die hinter dem Prädikat steht, kann also durch die Attribute des Objekts variabel parametrisiert werden. In Listing 5.32 nimmt der Konstruktor des Funktors eine Streamreferenz entgegen. Die mitgeführte Funktionalität in Form des Klammeroperators arbeitet dann mit dem im Konstruktor übergebenen Stream. Dieser kann also variabel sein, ohne dass der Klammeroperator einen weiteren Parameter entgegennimmt.

Listing 5.32. Variable Parametrisierung eines Prädikats

```
#include <iostream>
#include <fstream>
#include <list>
#include <algorithm>

class ausgabe
{
public:
    ausgabe( std::ostream &s ) : os(s) {}
    void operator()( int i ) const
    { os << i << std::endl; }
private:
    std::ostream &os;
};

int main()
{
    using namespace std;

    list<int> l1;

    l1.push_back( 1 );
    l1.push_back( 2 );
    l1.push_back( 3 );

    for_each( l1.begin(), l1.end(), ausgabe(cout) );
    ofstream datei( "test.txt" );
    for_each( l1.begin(), l1.end(), ausgabe(datei) );

    return 0;
}
```

Das Beispiel in Listing 5.32 schreibt auf die Standardausgabe und in eine Datei. Dabei wird die gleiche Funktorklasse verwendet.

Der nächste hier vorgestellte Algorithmus ist das Template `count_if()`. Er nimmt zwei Iteratoren entgegen, wie auch der Algorithmus `for_each()`. Das Prädikat, das als dritter Parameter übergeben wird, wird im Algorithmus als Bedingung gebraucht. Mit diesem Prädikat kann man den Algorithmus parametrisieren.

Listing 5.33. Der Algorithmus `count_if`

```
#include <iostream>
#include <list>
#include <algorithm>

bool test( int i )
{
    return (i % 2) == 0;
}

int main()
{
    using namespace std;

    list<int> l;

    l.push_back(  33 );
    l.push_back(  44 );
    l.push_back( 108 );
    l.push_back(  12 );

    int n = count_if( l.begin(), l.end(), test );
    cout << n << endl;

    return 0;
}
```

Auch dieses Prädikat kann als Klasse implementiert werden:

Listing 5.34. Funktorimplementierung des Prädikats

```
// ...
class test
{
public:
  bool operator()( int i )
  {
    return (i % 2) == 0;
  }
};
```

```
int main()
{
  // ...
  int n = count_if( l.begin(), l.end(), test() );
  // ...
}
```

Das nächste Beispiel zeigt ein parametrisierbares Prädikat. Es sollen einfach ganz bestimmte Werte in dem Container gezählt werden. Dazu müssen Werte verglichen werden. Das Prädikat nimmt deshalb zwei Argumente entgegen. Wenn einer dieser Werte aus dem Container geholt werden soll, muss der andere in einer anderen Form dem Prädikat übergeben werden. Dafür gibt es sogenannte Funktionsadapter, die eine Schicht zwischen dem Algorithmus und dem Prädikat bilden. Diese Adapter heißen `bind1st()` und `bind2nd()`. Sie nehmen als Argumente das zweiparametrige Prädikat und einen Wert für eines der beiden Prädikatargumente, im Fall von `bind1st()` ist es das erste und bei `bind2nd()` ist es das zweite. Als Rückgabe wird dem Algorithmus ein Prädikat geliefert, das nur einen Parameter enthält. Damit der Adapter das Rückgabeprädikat bilden kann, braucht er die drei Typendefinitionen `first_argument_type`, `second_argument_type` und `result_type`. Der Adapter ist ein Template und nutzt infolgedessen Templatetechniken, um das Prädikat für den Algorithmus zu bilden.

Listing 5.35. Adaptieren von Prädikaten

```
#include <iostream>
#include <list>
#include <algorithm>

class compare
{
public:
    typedef int first_argument_type;
    typedef int second_argument_type;
    typedef bool result_type;
    bool operator()( int i, int v ) const
    {
        return i == v;
    }
};

int main()
{
    using namespace std;
```

```
    list<int> l;

    l.push_back( 3 );
    l.push_back( 4 );
    l.push_back( 3 );
    l.push_back( 1 );

    int n = std::count_if( l.begin(),
                           l.end(),
                           bind2nd(compare(),3) );
    cout << n << endl;

    return 0;
}
```

Das Beispiel zählt die Elemente mit dem Wert drei. In diesem speziellen Fall hätte man auch den Adapter `bind1st()` verwenden können, denn es ist hier egal, welches Argument für das Prädikat `compare` aus dem Container stammt und welches der Vergleichswert ist.

Zur Vereinfachung kann das Prädikat von dem allgemeinen Prädikat `binary_function` abgeleitet werden, das als Templateklasse implementiert ist. Damit werden die drei Typen, die `bind2nd()` benötigt, definiert. In den spitzen Klammern der Templatedefinition stehen zuerst die Parametertypen der Operation und zuletzt der Rückgabetyp.

Listing 5.36. Standardkonforme Deklaration eines Prädikats

```
class compare
: public std::binary_function<int,int,bool>
{
public:
    bool operator()( int i, int v ) const
    {
        return i == v;
    }
};
```

Dieses Vorgehen ist besonders dann hilfreich, wenn man ein Prädikat als Template implementiert.

Listing 5.37. Standardkonforme Deklaration eines Prädikat-Templates

```
template <typename T>
class compare
: public std::binary_function<T,T,bool>
{
public:
    bool operator()( const T &v1, const T &v2 ) const
```

```
    {
        return v1 == v2;
    }
};
```

Das hier vorgestellte Prädikat `compare` wurde zum Zweck der Veranschaulichung der Techniken der STL entwickelt und muss nicht mehr selbst geschrieben werden, denn es gibt bereits etwas Entsprechendes in der STL. Die STL definiert einige zweiparametrige Prädikate, die eine gewisse Grundfunktionalität abdecken. Um Vergleiche durchzuführen sind das die folgenden Elemente:

```
template <typename T> struct equal_to;
template <typename T> struct not_equal_to;
template <typename T> struct greater;
template <typename T> struct less;
template <typename T> struct greater_equal;
template <typename T> struct less_equal;
```

Um das Beispiel also mit einem vordefinierten Prädikat auszustatten, braucht nur das entsprechende Template instanziiert zu werden:

Listing 5.38. Ein in der STL vordefiniertes Prädikat

```
// ...
int main()
{
  // ...
    int n = count_if( l.begin(), l.end(),
                      bind2nd(equal_to<int>(),3) );
    cout << n << endl;

    return 0;
}
```

Es gibt auch Funktoren, die ein Prädikat negieren:

```
template <typename P> unary_negate<P>  not1( const P& );
template <typename P> binary_negate<P> not2( const P& );
```

Um also in dem aktuellen Beispiel alle Elemente zu zählen, die ungleich drei sind, könnte man Folgendes schreiben:

Listing 5.39. Eine Modifikation des Prädikats

```
// ...
int n = count_if( l.begin(), l.end(),
                  not1(bind2nd(equal_to<int>(),3)));
// ...
```

Natürlich kann man ebenso das entsprechende Vergleichsprädikat verwenden:

Listing 5.40. Eine Umformung

```
// ...
int n = count_if( l.begin(), l.end(),
                  bind2nd(not_equal_to<int>(),3));
// ...
```

Verändernde Algorithmen

In den vorangegangenen Anwendungsbeispielen für Algorithmen und Funktoren wurden an den Datencontainern keine Änderungen durchgeführt. In diesem Abschnitt wird die Anwendung von verändernden Algorithmen gezeigt. Das nächste Beispiel demonstriert, wie die Werte eines Containers modifiziert werden können:

Listing 5.41. Verändernder Algorithmus `transform()`

```
#include <iostream>
#include <list>
#include <algorithm>
#include <functional>

class ausgabe
{
public:
    ausgabe( std::ostream &s ) : os(s) {}
    void operator()( int i ) const
    { os << i << std::endl; }
private:
    std::ostream &os;
};

int main()
{
    using namespace std;

    list<int> l;

    l.push_back( 1 );
    l.push_back( 2 );
    l.push_back( 3 );

    for_each( l.begin(), l.end(), ausgabe(cout) );
```

```
    transform( l.begin(), l.end(),
               l.begin(), negate<int>() );
    for_each( l.begin(), l.end(), ausgabe(cout) );

    return 0;
}
```

Mit dem Algorithmus `transform()` wird eine Sequenz aus einem Container in einen Zielcontainer kopiert und gleichzeitig transformiert. Dabei können die Ergebnisdaten auch unmittelbar in den Ursprungscontainer zurückkopiert werden. Die neuen Daten ersetzen dabei die alten, schon existierenden Daten. Der Zielcontainer muss also schon die entsprechende Größe zur Aufnahme der Daten haben.

Das Prädikat `negate<T>()` ist hier ein von der STL vordefiniertes. Zur Erinnerung sei darauf hingewiesen, dass es eine beliebige Funktion oder ein Funktionsobjekt sein kann, das als Prädikat verwendet werden kann. Die Algorithmen lassen sich auf Container anwenden, die die STL-konforme Schnittstelle besitzen. Sie müssen nicht aus der STL stammen:

Listing 5.42. Anwendung von `transform` auf einen C-String

```
// ...
using namespace std;
string str = "Ein Text";
transform( str.begin(), str.end(),
           str.begin(), toupper );
```

Wenn an einen Container die transformierten Daten angehängt werden sollen, kann das mit einem `back_inserter()` geschehen.

Listing 5.43. Transformation und Anhängen an neuen Container

```
// ...
list<int> l1;

l1.push_back( 1 );
l1.push_back( 2 );
l1.push_back( 3 );

for_each( l1.begin(), l1.end(), ausgabe(cout) );
list<int> l2;
transform( l1.begin(), l1.end(),
           back_inserter(l2), negate<int>() );
for_each( l2.begin(), l2.end(), ausgabe(cout) );
// ...
```

Der Algorithmus kann auch dazu verwendet werden, mehrere Sequenzen miteinander zu verrechnen. Das folgende Beispiel zeigt zwei gleich große Se-

quenzen, die über das Prädikat `power()` verrechnet werden. Dabei berechnet `power()` die Werte x^y für alle Werte x aus der ersten Sequenz und die Werte y aus der zweiten Sequenz.

Listing 5.44. Rechenoperationen auf Mengen mit dem Algorithmus `transform`

```
// ...
int power( int x, int y )
{
    int ret = 1;
    while( y-- ) ret *= x;
    return ret;
}

int main()
{
    using namespace std;

    list<int> l1;
    list<int> l2;

    l1.push_back( 11 );
    l1.push_back(  7 );
    l1.push_back(  2 );

    l2.push_back( 2 );
    l2.push_back( 3 );
    l2.push_back( 7 );

    transform( l1.begin(), l1.end(),
               l2.begin(), l1.begin(), power );
    for_each( l1.begin(), l1.end(), ausgabe(cout) );

    return 0;
}
```

Das Programm erzeugt die folgende Ausgabe:

```
121
343
128
```

Weitere modifizierende Algorithmen sind in der Copy-Familie die Templates `copy()`, `copy_backward()` und `copy_if()`. Das letztere benötigt ein Prädikat. Die beiden ersten benötigen kein Prädikat und haben ansonsten die gleichen Parameterlisten wie auch der Algorithmus `transform()`.

Listing 5.45. Der Algorithmus `copy`

```
// ...
int main()
{
    using namespace std;

    list<int> l;
    vector<int> v;

    l.push_back( 7 );
    l.push_back( 4 );
    l.push_back( 7 );

    v.resize( l.size() );

    copy( l.begin(), l.end(), v.begin() );
    for_each( v.begin(), v.end(), ausgabe(cout) );

    return 0;
}
```

Auch die Kopieralgorithmen setzen entweder vorhandenen Platz im Zielcontainer oder einen Inputiterator voraus.

Listing 5.46. Inputiterator durch `back_inserter`

```
// ...
int main()
{
    using namespace std;

    list<int> l;
    vector<int> v;

    l.push_back( 7 );
    l.push_back( 4 );
    l.push_back( 7 );

    copy( l.begin(), l.end(), back_inserter(v) );
    for_each( v.begin(), v.end(), ausgabe(cout) );

    return 0;
}
```

Es gibt noch weitere Algorithmen in der STL, die alle ähnlich zu verwenden sind wie die vorgestellten. Die Implementierungen dieser Algorithmen sind im Allgemeinen sehr einfach, was eine eigene Implementierung eines Algo-

rithmus für STL-Container erleichtert. Um die Art der Implementierung von Algorithmen zu demonstrieren, soll eine eigene Variante von `for_each<>()` implementiert werden:

Listing 5.47. Eigene Implementierung eines Algorithmus

```
template<typename I, typename P>
void my_for_each( I b, I e, P p )
{
  for( I i = b; i != e; ++i )
    p( *i );
}
```

Diese eigene Implementierung ist ebenso einfach wie die in der STL. Die eigene Implementierung `my_for_each<>()` funktioniert mit allen vorangegangenen Beispielen, in denen `for_each<>()` verwendet wurde.

5.2.7 Die Stringklasse `std::string`

Die Stringklasse `std::string` in der Standardbibliothek verhält sich nach außen wie ein Container aus der STL obwohl sie eigentlich nicht Bestandteil der STL ist. Sie hat die gleiche Iteratorschnittstelle und basiert auf Templates. Mit der Stringklasse wurde das Prinzip in der Konstruktion von Containern in weitere Bereiche der Standardbibliothek übernommen. Das folgende Beispiel soll zeigen, dass es sich bei der Stringklasse um einen weiteren Containertyp der bekannten STL-Bauart handelt.

Listing 5.48. Iteratorzugriff auf ein `std::string`-Objekt

```
#include <iostream>
#include <string>

int main()
{
    using namespace std;

    string s( "Test" );

    cout << s << endl;

    for( string::iterator it = s.begin();
         it != s.end(); ++it )
        cout << *it << " ";

    return 0;
}
```

Die Klasse `std::string` verwaltet Zeichenketten und bietet die notwendigen Methoden, um dem Benutzer den Umgang mit den Zeichenketten möglichst einfach zu gestalten. Dazu gehören einfach zu benutzende Konstruktoren:

Listing 5.49. Verschiedene Konstruktoren der Stringklasse

```
#include <iostream>
#include <string>

int main()
{
    using namespace std;

    string str1( "Ostern" );
    string str2 = "in Madeira";
    string str3;
    string str4( str1, 0, 5 );
    string str5( str2, 6, 2 );
    string str6 = str4 + str5;

    cout << str6 << endl;

    return 0;
}
```

Die Konstruktoren von `str4` und `str5` schneiden Teilstrings aus den schon existierenden Strings heraus. Dabei ist zu beachten, dass keine Bereichsüberschreitung stattfinden kann, denn diese wirft die Ausnahme `out_of_range`. Damit man einen Teilstring bis zu seinem Ende kopieren kann, ohne die genaue Länge des Strings zu wissen, gibt es eine Konstante, die man anstatt eines genauen Wertes eintragen kann. Das nächste Beispiel zeigt die Anwendung dieser Konstante `string::npos` im Konstruktoraufruf von `str3`.

Listing 5.50. Anwendung von `string::npos`

```
#include <iostream>
#include <string>

int main()
{
    using namespace std;

    char *p = "Schnapsflasche";
    string str1( "Polyamid" );
    string str2( p, 7 );
```

```
    string str3( str1, 6, string::npos );
    string str4( 2, 'e' );

    cout << (str2 + str3 + str4) << endl;

    return 0;
}
```

Die Klasse `string` hat außerdem einen Konstruktor, der als Membertemplate implementiert ist und zwei Iteratoren entgegennimmt. Er liest über die Iteratoren den Inhalt des entsprechenden Containers in dem Intervall, das durch die Iteratoren angegeben ist, aus[12].

Listing 5.51. Initialisierung aus einer Containersequenz

```
#include <iostream>
#include <string>
#include <list>

int main()
{
    using namespace std;

    list<char> l;
    l.push_back( 'T' );
    l.push_back( 'e' );
    l.push_back( 'x' );
    l.push_back( 't' );

    string s( l.begin(), l.end() );

    cout << s << endl;

    return 0;
}
```

Die STL-Liste kann durch einen beliebigen anderen Container ersetzt werden[13].

Der Stringklasse `string` liegt ein Template zugrunde, das mit einem Basistyp, einer Klasse, die das Verhalten des Strings definiert, und einem Allokatortyp parametrisiert wird.

[12] Natürlich muss der Inhaltstyp des Containers zum Basistyp des Strings passen.

[13] Leider lässt sich das Beispiel nicht mit jedem aktuellen Compiler übersetzen. Dazu muss er Templatemember innerhalb von Klassentemplates unterstützen. Mit dem STL-Vektor schaffen es alle getesteten Compiler.

```
template < class E,
 class T = char_traits<E>,
 class A = allocator<E> >
class basic_string;
```

Da die letzten beiden Templateparameter mit Vorgaben belegt sind, müssen sie nicht zwingend angegeben werden und der `string` ist folgendermaßen definiert:

```
typedef basic_string<char> string;
```

Damit steht dem Entwickler die Definition eigener Stringklassen offen. Er kann die unterschiedlichen Aspekte – Traits und Allokator – variieren oder einen anderen Basistyp für den String wählen. Das folgende Beispiel zeigt einen auf `wchar_t` basierenden String.

Listing 5.52. Definition einer auf `wchar_t` basierenden Stringklasse

```
#include <iostream>
#include <string>

typedef std::basic_string< wchar_t > wstring;

std::ostream& operator<<( std::ostream &os,
                          const wstring &ws )
{
    for( wstring::const_iterator it = ws.begin();
         it != ws.end(); ++it )
        os << (char)(*it);
    return os;
}

int main()
{
    wstring ws1 = L"Ein Test";

    std::cout << ws1 << std::endl;

    return 0;
}
```

Der zweite Templateparameter betrifft den Aspekt des Verhaltens des Strings. Man kann einen eigenen Typ für den `char_traits`-Parameter schreiben. Dabei müssen bestimmte statische Methoden zur Verfügung gestellt werden, die die Stringklasse benutzt und deren Implementierungen das Verhalten des Strings festlegen. Im nachfolgenden Beispiel wird eine Klasse `MyCharTraits` definiert, die für den Vergleich der Strings keinen Unterschied zwischen großen und kleinen Buchstaben macht. Der vordefinierte

`std::string` ist casesensitiv. Der String, der im nachfolgenden Beispiel definiert werden soll, ist gerade nicht casesensitiv. Demonstriert wird das anhand zweier Container, die Strings enthalten. Für den Container wurde das Template `std::set<T>` instanziiert, das intern als Binärer Baum organisiert ist und seine Elemente vorsortiert. Zur Vorsortierung werden von dem Template `set` Vergleiche zwischen Strings durchgeführt. Diese haben für die Klasse `MyString` ein anderes Verhalten als für `std::string`.

Listing 5.53. Definition einer Stringklasse mit anderen Eigenschaften

```
#include <iostream>
#include <string>
#include <cctype>
#include <set>

struct MyCharTraits : public std::char_traits<char>
{
  typedef unsigned int  size_t;
  static bool eq( const char& a, const char& b )
  {
    return std::tolower( a ) == std::tolower( b );
  }
  static bool lt( const char& a, const char& b )
  {
    return std::char_traits<char>::lt(std::tolower(a),
                                      std::tolower(b));
  }
  static int compare( const char *a,
                      const char *b, size_t n )
  {
    int ret = 0;
    for( size_t i = 0; i < n && ret == 0; ++i )
    {
      if(      lt( b[i], a[i] ) ) ret =  1;
      else if( lt( a[i], b[i] ) ) ret = -1;
    }
    return ret;
  }
};

typedef std::basic_string<char,MyCharTraits> MyString;
std::ostream& operator<<( std::ostream &os,
                          const MyString &s )
{ return os << s.c_str(); }
```

```
typedef std::set<std::string> BTree1;
typedef std::set<MyString>    BTree2;

int main()
{
    using namespace std;

    BTree1 t1;
    BTree2 t2;

    t1.insert( "arabisch" );
    t1.insert( "Arabien" );
    t1.insert( "afrikanisch" );
    t1.insert( "Afrika" );

    for( BTree1::iterator it1 = t1.begin();
         it1 != t1.end(); ++it1 )
        cout << *it1 << endl;

    cout << "**********" << endl;

    t2.insert( "arabisch" );
    t2.insert( "Arabien" );
    t2.insert( "afrikanisch" );
    t2.insert( "Afrika" );

    for( BTree2::iterator it2 = t2.begin();
         it2 != t2.end(); ++it2 )
        cout << *it2 << endl;

    return 0;
}
```

Die Ausgabe des Programms:

```
Afrika
Arabien
afrikanisch
arabisch
**********
Afrika
afrikanisch
Arabien
arabisch
```

Der Typ, der als Traits-Klasse fungieren soll, muss außer `eq()`, `lt()` und `compare()` noch viele andere Methoden zur Verfügung stellen. Da es das Beispiel sprengen würde, alle Methoden vollständig zur Verfügung zu stellen, wurden für die Klasse `MyCharTraits` die restlichen Methoden aus der Klasse `std::char_traits<char>` durch Vererbung übernommen. Der Traits-Typ muss nicht von dieser Elternklasse abgeleitet werden. Da es sich um Templatetechniken handelt, muss nur die Morphologie der Schnittstelle des Typs stimmen.

5.2.8 Pseudocontainer für optimiertes Rechnen

Die Standardbibliothek enthält Klassen, um Arrays und Matrizen darzustellen, auf die Rechenoperationen optimiert durchgeführt werden können. Grundsätzlich sind natürlich auch die STL-Container dazu in der Lage, dass man mit ihnen Verrechnungen ganzer Arrays von Werten implementiert oder Matrizenoperationen durchführt. Das eigentliche Designziel des Templates `valarray<T>` war aber die Optimierbarkeit, woraus eine Existenzberechtigung neben den STL-Containern in der Standardbibliothek begründet werden kann. Die Schnittstelle von `valarray<T>` bietet deshalb auch nicht so viele Methoden und Möglichkeiten wie die Konkurrenz aus der STL. Es werden nur die notwendigsten geboten und diese auch nur so, dass sie in ihrer Implementierung optimierbar bleiben. Diese Implementierung ist allerdings abhängig vom Hersteller der Bibliothek bzw. des Entwicklungssystems. Durch verschiedene Spezialisierungen des Templates kann der Hersteller der Bibliothek das `valarray<T>` für ganz bestimmte Inhaltstypen optimieren[14].

Grundsätzlich ist ein `valarray<T>` organisiert wie ein Vektor.

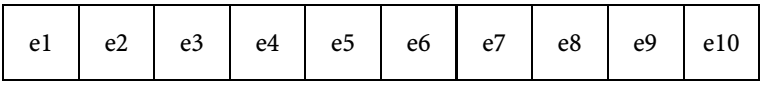


Abbildung 5.5. Einfache Organisation eines Valarrays

Die Elemente liegen direkt benachbart im Speicher, was einen indizierten Zugriff erlaubt. Iteratoren kennt das `valarray<T>` nicht, dafür die Methode `size()`, die ein komfortables Abfragen der Elementanzahl erlaubt. Im Konstruktor muss die Größe des Arrays angegeben werden.

[14] Insbesondere die ganzzahligen Standarddatentypen lassen gute Optimierungen einfach erreichen. Durch spezielle Anpassung des Valarrays an die Hardware lässt sich beispielsweise eine Unterstützung der MMX- und SSE-Befehlssätze bei den Intelprozessoren erreichen.

Listing 5.54. Einfache Verwendung eines Valarrays

```
#include <iostream>
#include <valarray>

void print_out( const std::valarray<int> &va )
{
    for( unsigned i = 0; i < va.size(); ++i )
        std::cout << va[i] << std::endl;
}

int main()
{
    std::valarray<int> v(10);

    for( unsigned i = 0; i < 10; ++i )
        v[i] = 10 * (i+1);

    print_out( v );

    return 0;
}
```

Es ist allerdings auch möglich, die Größe von einem `valarray<T>` nachträglich zu ändern. Dazu gibt es die Methode `resize()`:

```
void resize(size_t sz, T c = T())
```

Sie vergrößert das Array auf die Größe, die im ersten Parameter angegeben ist, und initialisiert alle Elemente mit dem Wert des zweiten Parameters[15]. Da der zweite Parameter mit einem Vorgabewert belegt ist, muss er nicht angegeben werden. In diesem Fall fügt der Compiler einen Aufruf des Standardkonstruktors des Inhaltstyps an der Stelle des zweiten Parameters ein.

Listing 5.55. Vergrößerung des Valarrays durch Aufruf von `resize()`

```
#include <iostream>
#include <valarray>

void print_out( const std::valarray<int> &va )
{
    for( unsigned i = 0; i < va.size(); ++i )
        std::cout << va[i] << std::endl;
```

[15] Achtung! In diesem Punkt reagiert die Bibliothek des weit verbreiteten MS Visual C++ 7.1 nicht standardkonform. Dort werden nur die letzten Elemente, die neu hinzugekommen sind, mit dem übergebenen Wert initialisiert. Zum Vergleich dazu: [ISO-C++] Abschnitt 26.3.2.7 [lib.valarray.members].

```
}

int main()
{
    std::valarray<int> v(10);

    for( unsigned i = 0; i < 10; ++i )
        v[i] = 10 * (i+1);

    print_out( v );

    std::cout << "***" << std::endl;

    v.resize( 11 );

    print_out( v );

    return 0;
}
```

Das `valarray<T>` hat Methoden, um den Minimalwert, den Maximalwert und die Quersumme des ganzen Arrays zu bestimmen: `min()`, `max()` und `sum()`. Voraussetzung für die Anwendung dieser Methoden ist, dass für den Inhaltstyp die entsprechenden Operatoren („<“ und „+“) definiert sind. Interessant wird die Anwendung des Templates `valarray<T>` aber, wenn Operationen auf das ganze Array oder zwischen Arrays stattfinden sollen. Alle arithmetischen und alle Bitoperatoren sind für das `valarray<T>` überladen. Es definiert auch alle „computed assignments“, d. h. sowohl alle Mischformen der arithmetischen Operatoren mit der Zuweisung[16] als auch alle Bitoperatoren mit der Zuweisung zwischen Arrays gleicher Größe. Auch hier gilt wieder die Voraussetzung, dass der Inhaltstyp die entsprechenden Operationen definiert. Der Grund, warum möglichst diese Mischformen zwischen Zuweisung und arithmetischer Operation benutzt werden sollten und nicht die Operation in Reinform, ist, dass eine reine Operation ein Ergebnis hat, das auf dem Stack angelegt werden muss, um später umkopiert zu werden. In der Mischform spart man sich das anonyme Objekt mit all dem Speicherverbrauch und Rechenaufwand. Wenn solchen Operatoren `valarray<T>`-Objekte übergeben werden, werden alle Elemente der Arrays miteinander verrechnet. Den Operationen kann aber auch ein `valarray`-Objekt und ein Einzelwert übergeben werden. In dem Fall wird der Einzelwert mit allen Arraywerten verrechnet.

[16] +=, -=, *= ...

Listing 5.56. Arithmetische Operationen auf Valarrays

```
#include <iostream>
#include <valarray>

void print_out( const std::valarray<int> &va )
{
    for( unsigned i = 0; i < va.size(); ++i )
        std::cout << va[i] << " ";
    std::cout << std::endl;
}

int main()
{
    int a[] = { 5, 4, 1, 2 };
    std::valarray<int> v1(a,4);
    std::valarray<int> v2(2,4);
    v2[0] =  1;
    v2[1] =  0;

    print_out( v1 );
    print_out( v2 );
    v1 += v2;
    print_out( v1 );
    v1 += 3;
    print_out( v1 );

    return 0;
}
```

Es sind noch unäre Operatoren definiert, die wiederum auf jedes Einzelelement angewendet werden können: („+“, „-“, „~“ und „!“). Außerdem stehen trigonometrische und andere mathematische Funktionen zur Verfügung, die auf `valarray`-Objekte angewandt werden können.

```
template<class T> valarray<T> abs( const valarray<T>& );
template<class T> valarray<T> acos( const valarray<T>& );
template<class T> valarray<T> asin( const valarray<T>& );
template<class T> valarray<T> atan( const valarray<T>& );
template<class T> valarray<T> atan2( const valarray<T>&,
                                     const valarray<T>& );
template<class T> valarray<T> atan2( const valarray<T>&,
                                     const T&  );
template<class T> valarray<T> atan2( const T&,
                                     const valarray<T>& );
```

```
template<class T> valarray<T> cos( const valarray<T>& );
template<class T> valarray<T> cosh( const valarray<T>& );
template<class T> valarray<T> exp( const valarray<T>& );
template<class T> valarray<T> log( const valarray<T>& );
template<class T> valarray<T> log10( const valarray<T>& );
template<class T> valarray<T> pow( const valarray<T>&,
                                   const T& );
template<class T> valarray<T> pow( const T&,
                                   const valarray<T>& );
template<class T> valarray<T> sin( const valarray<T>& );
template<class T> valarray<T> sinh( const valarray<T>& );
template<class T> valarray<T> sqrt( const valarray<T>& );
template<class T> valarray<T> tan( const valarray<T>& );
template<class T> valarray<T> tanh( const valarray<T>& );
```

Die beiden Methoden `shift()` und `cshift()` verschieben die Werte im Array um die Anzahl der Positionen, die im Parameter angegeben wurde. Dabei „rotiert" die Methode `cshift()`. Das heißt, dass Elemente, die auf der einen Seite aus dem Array herausfallen, auf der anderen Seite wieder hereingeschoben werden. Bei der Methode `shift()` gehen die Elemente verloren. Die neu entstandenen Plätze auf der anderen Seite werden mit Nullelementen[17] gefüllt.

Neben diesen Operationen auf das Array können eigene Funktionen in Form eines Funktionsobjekts definiert werden, die dann mit der Methode `apply()` auf alle Arrayelemente angewendet werden können. Diese Methode erzeugt ein Ergebnisarray.

Listing 5.57. Mengenoperationen auf Valarrays

```
#include <iostream>
#include <valarray>

void print_out( const std::valarray<int> &va )
{
    for( unsigned i = 0; i < va.size(); ++i )
        std::cout << va[i] << " ";
    std::cout << std::endl;
}

int loesche_gerade_zahlen( int i )
{
    return ( i % 2 ) ? i : 0;
}

int main()
```

[17] Elemente, die mit dem Standardkonstruktor erstellt wurden.

```
{
    using namespace std;
    int a[] = { 5, 4, 1, 2 };
    valarray<int> v1(a,4);

    print_out( v1 );

    valarray<int> v2 = v1.apply(loesche_gerade_zahlen);

    print_out( v2 );

    return 0;
}
```

`slice`

Ein starkes Feature des Klassentemplates `valarray<T>` entsteht im Zusammenwirken mit der Klasse `slice`. Diese Klasse stellt eine Abstraktion dar, von der aus man das Array als eine Matrize betrachten kann. Die Klasse `slice` ist dabei ein „Mengenindex". Indem ein `slice`-Objekt der Liste als Indexwert übergeben wird, erhält man als Ergebnis ein Objekt der Klasse `slice_array<T>`, das auch wieder in ein `valarray<T>` konvertiert werden kann. Das Template `slice_array<T>` ist ein Hilfstemplate im Zusammenhang mit dem Zugriff auf Matrizen, denen ein `valarray<T>` zugrunde liegt. Objekte von `slice_array` muss man nie selbst instanziieren, weshalb die Konstruktoren auch alle privat sind. Die Instanziierung wird von `valarray` übernommen. Sie geschieht innerhalb des überladenen Indexoperators `[]` von `valarray<T>`. Ein Objekt der Klasse `slice` stellt praktisch einen Indexwert dar, der den Indexklammern `[]` des `valarray`-Objekts übergeben werden kann.

Listing 5.58. Submengenbildung durch Indizierung mit `slice`-Objekten

```
#include <iostream>
#include <valarray>

std::ostream & operator<<( std::ostream &os,
                     const std::valarray<int> &va )
{
    for( int i = 0; i < va.size(); ++i )
        os << va[i] << "  ";
    return os;
}

int main()
{
    using namespace std;
```

```
    int a[] = { 10, 11, 12, 20, 21, 22,
                30, 31, 32, 40, 41, 42 };
    valarray<int> v(a,3*4);

    slice z1( 0, 3, 1 );
    slice z2( 3, 3, 1 );
    slice z3( 6, 3, 1 );
    slice z4( 9, 3, 1 );
    slice s1( 0, 4, 3 );
    slice s2( 1, 4, 3 );
    slice s3( 2, 4, 3 );

    valarray<int> zeile1 = v[z1];
    valarray<int> zeile2 = v[z2];
    valarray<int> zeile3 = v[z3];
    valarray<int> zeile4 = v[z4];

    valarray<int> spalte1 = v[s1];
    valarray<int> spalte2 = v[s2];
    valarray<int> spalte3 = v[s3];

    cout << zeile1 << endl;
    cout << zeile2 << endl;
    cout << zeile3 << endl;
    cout << zeile4 << endl;

    cout << endl;

    cout << spalte1 << endl;
    cout << spalte2 << endl;
    cout << spalte3 << endl;

    return 0;
}
```

Die Ausgabe des Programms:

```
10 11 12
20 21 22
30 31 32
40 41 42

10 20 30 40
11 21 31 41
12 22 32 42
```

Die nachfolgende grafische Darstellung zeigt den linearen Aufbau des `valarray`-Objektes:

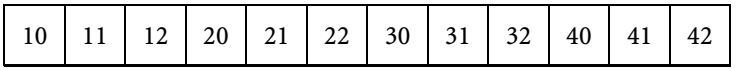


Abbildung 5.6. Der lineare Aufbau eines Valarray-Objekts

Diese Oranisationsform erinnert an einen Vektor aus der STL. Unter Zuhilfenahme des `slice`-Objektes kann man eine Matrizensicht erreichen:

10	11	12
20	21	22
30	31	32
40	41	42

Abbildung 5.7. Die Matrizendarstellung

Der hier wesentliche Konstruktor von `slice` nimmt drei Argumente: den Startpunkt, die Anzahl der Elemente und die Sprungweite über das Array.

```
slice(size_t start, size_t length, size_t stride);
```

Im vorangegangenen Beispiel wurde die erste Zeile der Matrix dadurch indiziert, dass von der Position 0 aus drei Elemente in der Schrittweite 1 abgefragt wurden.

```
slice z1( 0, 3, 1 );
```

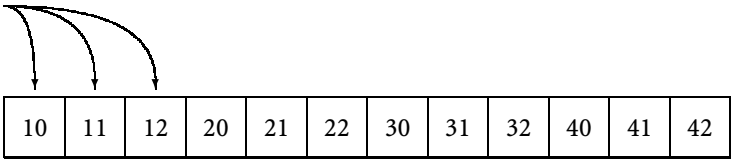


Abbildung 5.8. Die Elementauswahl durch ein `slice`-Objekt

Die erste Spalte kommt dadurch zustande, dass von der Position 0 aus vier Elemente mit der Schrittweite 3 indiziert wurden.

```
slice s1( 0, 4, 3 );
```

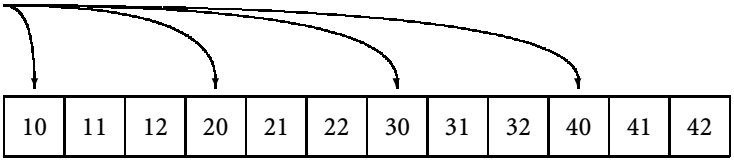


Abbildung 5.9. Eine weitere Elementauswahl durch ein `slice`-Objekt

Die Klasse `slice` hat noch einen Standard- und einen Kopierkonstruktor, so dass ein `slice`-Objekt auch leer initialisiert und kopiert werden kann. Außerden kann die Auswahl, die durch ein `slice_array` repräsentiert wird, auch verändert werden.

Listing 5.59. Veränderung der Werte einer `slice`-Auswahl

```
// ...
v[slice(1,4,3)] = 0;
// ...
```

Diese Zeile verändert das zugrunde liegende `valarray` in der Weise, dass alle Elemente, die durch das `slice`-Objekt indiziert wurden, mit dem Wert `0` belegt werden. Ein Objekt der Klasse `slice_array` speichert also die Werte nicht direkt, sondern enthält Verweise auf die Elemente, die in dem `valarray`-Objekt stehen. Die Klasse `slice_array` wurde also nur dazu gemacht, die Ergebnismengen aus einer Indizierung eines `valarray`-Objekts zu repräsentieren. Wenn eine Ergebnismenge separat gehalten werden soll, also physisch aus dem `valarray`-Objekt herauskopiert werden soll, kann man sie wieder in einem `valarray` speichern. Die Klasse `valarray` hat dazu einen Konstruktor, der einen Parameter vom Typ `slice_array` entgegennimmt.

Die gezeigten Beispiele beziehen sich auf eine zweidimensionale Matrix. Alle besprochenen Funktionen lassen sich aber auch auf mehrdimensionale Matrizen anwenden.

`gslice`

Mit den `slice`-Objekten können Matrizensichten auf Arrays definiert werden. Sie verhalten sicch damit als logische Matrizen. Aus einer solchen Matrix können auch Bereiche als Submatrizen separiert werden. Dazu gibt es die Klasse `gslice`, die eine ebensolche Abstraktion darstellt, wie die der `slice`-Klasse.

Listing 5.60. Eine Submatrix mit `gslice`

```
#include <iostream>
#include <valarray>

std::ostream & operator<<( std::ostream &os,
                    const std::slice_array<int> &sa )
{
```

```
    std::valarray<int> va( sa );
    for( int i = 0; i < va.size(); ++i )
        os << va[i] << "\t";
    return os;
}

int main()
{
    using namespace std;
    int a[] = { 10, 11, 12, 20, 21, 22,
                30, 31, 32, 40, 41, 42 };
    valarray<int> v(a,3*4);

    valarray<size_t> laengebreite(2);
    valarray<size_t> schritte(2);

    laengebreite[0] = 2;
    laengebreite[1] = 1;
    schritte[0] = 3;
    schritte[1] = 1;
    gslice gs(7, laengebreite, schritte );

    valarray<int> e = v[gs];

    cout << e[slice(0,2,1)] << endl;
    cout << e[slice(2,2,1)] << endl;

    return 0;
}
```

Die Ausgabe des Programms:

```
31  32
41  42
```

Aus der 3 × 4-Matrix wurde eine 2 × 2-Matrix herausgegriffen.

10	11	12
20	21	22
30	**31**	**32**
40	**41**	**42**

Abbildung 5.10. Die 2 × 2-Submatrix aus der 3 × 4-Matrix

Die Anwendung eines `gslice`-Objektes ermöglicht also eine Matrizensicht auf linear abgelegte Daten in einem `valarray`. Kombiniert mit der Optimierbarkeit der zugrunde liegenden Operationen im `valarray` empfiehlt sich die Matrizendarstellung mit `gslice` beispielsweise für die Grafikprogrammierung oder für numerisch aufwändige mathematische Berechnungen.

5.2.9 Autopointer

Es gibt eine Smart-Pointer-Implementierung in der Standardbibliothek, die sich jedoch von der in diesem Buch beschriebenen Implementierung im Abschnitt 4.1.15 etwas unterscheidet. Das Klassentemplate `auto_ptr<T>` ist dazu gedacht, einen dynamisch allokierten Zeiger des Templateparametertyps aufzunehmen und das referenzierte Objekt zu zerstören, wenn der Autopointer selbst gelöscht wird. Wenn allerdings ein Autopointer kopiert wird, ist es die Kopie, die bei ihrem eigenen Entfernen aus dem Speicher das referenzierte Objekt zerstört. Die „Eigentumsbeziehung" des Autopointers geht also an die Kopie über. Für den Umgang mit dem Autopointer ist also das genaue Verständnis dieser Eigentumsbeziehung notwendig.

Das folgende Beispiel zeigt die Verwendung des Templates `auto_ptr<T>`.

Listing 5.61. Automatische Freigabe des dynamisch instanziierten Objekts

```
#include <iostream>
#include <memory>

class X
{
public:
    X()  { std::cout << "X::X()"  << std::endl; }
    ~X() { std::cout << "X::~X()" << std::endl; }
};

int main()
{
    std::auto_ptr<X> p( new X() );

    return 0;
}
```

Da der Autopointer `p` die „Verantwortung" für das dynamisch allokierte Objekt der Klasse `X` hat, löscht es dieses, wenn er selbst aus dem Speicher entfernt wird. Diese Eigentumsbeziehung kann auch aufgelöst werden. Dazu gibt es die Methode `release()`. Sie löst die Eigentumsbeziehung des Autopointers zu dem enthaltenen Objekt auf und gibt dessen Zeiger zurück.

Listing 5.62. Lösen des Objekts aus der Verantwortung des Autopointers

```
// ...
int main()
{
    std::auto_ptr<X> p( new X() );

    X *xp = p.release();

    delete xp;

    return 0;
}
```

Die Ausgabe dieses Programms ist die gleiche wie die Ausgabe des vorherigen. Das Objekt der Klasse `X` wird hier explizit mit `delete` gelöscht. Würde es später noch durch den Autopointer `p` gelöscht werden, würde das zu einem Laufzeitfehler führen. Mit der Methode `reset()` wird das Löschen des durch den Autopointer gehaltenen Objekts vorgezogen.

Listing 5.63. Vorgezogenes Löschen des Objekts am Autopointer

```
// ..
int main()
{
    using std::auto_ptr;

    auto_ptr<X> p( new X() );

    p.reset();

    p = auto_ptr<X>(new X());

    return 0;
}
```

Die Zuweisung eines neuen Autopointerobjekts zieht einen Aufruf von `reset()` nach sich, so dass der folgende Code fast die gleiche Semantik hat wie der vorhergehende; mit dem einen Unterschied, dass das neue `X`-Objekt instanziiert wird, bevor das alte gelöscht wird.

Listing 5.64. Automatisches Reset durch Neuzuweisung des Autopointers

```
// ...
int main()
{
    using std::auto_ptr;
```

```
    auto_ptr<X> p( new X() );

    p = auto_ptr<X>(new X());

    return 0;
}
```

Durch die überladenen Zugriffsoperatoren * und -> kann auf das enthaltene Objekt zugegriffen werden.

Listing 5.65. Die Operatoren * und -> am Autopointer

```
#include <iostream>
#include <memory>

class X
{
public:
    X()  { std::cout << "X::X()"  << std::endl; }
    ~X() { std::cout << "X::~X()" << std::endl; }

    void f() { std::cout << "X::f()" << std::endl; }
};

int main()
{
    using namespace std;

    auto_ptr<X> p1( new X() );
    auto_ptr<int> p2( new int() );

    p1->f();

    *p2 = 42;

    cout << *p2 << endl;

    return 0;
}
```

Der Autopointer `std::auto_ptr<>` wird gerne dort verwendet, wo man eine dynamische Objektinstanziierung durchführen muss, weil das Objekt für den Stack zu groß ist. Man möchte aber mit dem Objekt etwa so umgehen, wie es auch mit automatisch instanziierten getan werden kann. In diesem Zusammenhang ergibt der Autopointer Sinn. Ein anderer sehr wichtiger Anwendungsbereich ist die Sicherung dynamisch allokierter Objekte in einem

Algorithmus, der Ausnahmen werfen kann. Wenn Ausnahmen geworfen werden, wird der Stack bis zu der fangenden Stelle aufgeräumt. Das heißt, dass alle automatisch allokierten Objekte aufgeräumt werden. Ihr Destruktor wird aufgerufen. Hat man jedoch etwas dynamisch allokiert, muss man sich um das Löschen selber kümmern. Dynamisch mit `new` instanziierte Objekte können durch den Exceptionmechanismus nicht gelöscht werden. Wenn nun durch den Mechanismus der Stackbereinigung die lokalen Zeiger auf dynamisch allokierte Objekte verloren gehen, hat man Speicherlöcher im System – Objekte und Speicherbereiche, die zwar allokiert aber unerreichbar sind, also auch nicht mehr freigegeben werden können. Wenn man nun Autopointer anstatt herkömmliche Zeiger verwendet, dann werden die Autopointer, wenn sie aus dem Stack entfernt werden, auch die Objekte korrekt löschen. Da der Copy-Konstruktor-Aufruf[18] des Autopointers dazu führt, dass die Kopie das Eigentumsrecht am Objekt bekommt[19], lässt sich das Autopointerprinzip über mehrere Stufen über Funktionsaufrufe und Funktionsrückgaben anwenden. Eine C++-Bibliothek braucht also einen solchen Zeigertyp, wenn im Ausnahmefall Exceptions geworfen werden sollen.

[18] Siehe das Unterkapitel über das Verhalten des Copy-Konstruktors 2.2.21 auf Seite 87.
[19] Durch den Aufruf der Methode `release()` am Ursprungsobjekt.

6 Das Softwareprojekt mit C++

Ein Softwareprojekt mit C++ unterliegt meistens ganz spezifischen technischen Beschränkungen, die unabhängig zu den fachlichen Aspekten durch die Entwickler beherrscht werden müssen. Da ist zum einen die Aufteilung in Module[1], die ein grundsätzliches Problem darstellt. Einerseits muss man zur Beherrschung der Masse den Quellcode in Module aufteilen. Damit werden die Dateien kleiner und übersichtlicher, an denen die einzelnen Entwickler arbeiten. Andererseits möchte man die Compilezeiten kurz halten und den Überblick über das Gesamtprojekt nicht durch zu viele Einzelmodule erschweren. Dabei kann eine Modularisierung genau diese beiden Konsequenzen haben. Die Compilezeit eines Projektes steigt enorm an und der Entwickler hat Schwierigkeiten, die richtige Quellcodedatei für seine aktuelle Aufgabe zu finden.

Zu diesen grundsätzlichen Problemen gesellen sich noch andere Aspekte, die ebenfalls bedacht werden müssen: Da ist zum einen das Anwachsen des Codes durch den Einsatz von Templatetechniken, wenn diese nicht beherrscht werden. Zum anderen gibt es ähnliche Code-vergrößernde Mechanismen beim Einsatz von OO-Techniken, wenn Compiler und Linker nicht optimal aufeinander abgestimmt sind. Zusätzlich führen beide Aspekte zu einem weiteren Anwachsen der Compile- und Linkzeiten.

Es gibt also einiges zu bedenken, wenn die genannten Auswirkungen auf ein Projekt nicht akzeptabel sind.

6.1 Modularisierung eines C++ - Projekts

Die Aufteilung eines C++-Projektes in Module geschieht im Einzelfall nach sehr unterschiedlichen fachlichen Kriterien. Mit der Modularisierung wird aber immer eine Aufteilung in handliche kleine Codeteile angestrebt, die möglichst fachliche Einheiten bilden. Wenn einzelne fachliche Einheiten zu groß werden, wird eine weitere Aufteilung vorgenommen. In Bibliotheksprojekten ist es häufig so, dass jede einzelne Funktion eine extra Datei bekommt. Wenn der Umfang der Bibliothek vordefiniert ist oder einfach nur sehr gut dokumentiert, lassen sich dann die einzelnen Teile anhand der Dateinamen finden (das wiegt den Nachteil der vielen Dateien auf). Außerdem kommt

[1] Unter Modulen versteht man in C++ einfach die Quellcodedateien, die Implementierung enthalten. Zu ihnen gehört meistens auch eine Headerdatei. Bei manchen anderen Programmiersprachen wie zum Beispiel Modula-2 ist der Begriff „Modul“ etwas anders belegt. Er bezeichnet ein Stück Software mit zur Entwicklungszeit auslesbarer Schnittstelle.

man mit einer solchen totalen Aufsplittung älteren Entwicklungssystemen entgegen, die nur dann selektiv linken können, wenn jede Funktion in einer extra Objektdatei liegt. Selbst mit moderneren Linkern stellt dieses Thema oft noch ein Problem dar. Allen Aufteilungen in Module ist gemeinsam, dass Headerdateien die deklaratorische Klammer der Module sind. Die Elemente, die in den Modulen definiert sind, sind in Headerdateien deklariert und können so in anderen Modulen verwendet werden. Dazu wird die Headerdatei in dem Modul inkludiert, in dem das deklarierte Element verwendet werden soll. Die wesentliche Frage für die Modularisierung eines Projekts ist die Gestaltung der Headerdateien und der Umgang mit diesen. Wenn für jedes Modul eine Headerdatei geschrieben wird, kann das für Projekte mit zahlreichen Modulen bedeuten, dass sehr viel inkludiert werden muss. Deshalb fasst man oft die Deklarationen mehrerer Module zu einer Headerdatei zusammen. Die Entscheidung für oder gegen die Zusammenfassung von Deklarationen ist jedoch nur ein Aspekt des Umgangs mit Headerdateien.

6.1.1 Der Umgang mit Headerdateien

Headerdateien nehmen für gewöhnlich Präprozessordefines, Deklarationen und Typendefinitionen auf. Wie im Abschnitt über die Templates 4.1 näher erläutert wird, werden auch Templates in Headerdateien aufgenommen. Indem sich die deklarativen Teile des Quellcodes in Headerdateien befinden, können diese in mehreren Modulen inkludiert werden, um die Deklarationen dort bekannt zu machen. Die Deklarationen stellen so eine Art Vertrag dar, an den sich der funktionale Code zu halten hat. Der Compiler überprüft die Einhaltung des Vertrags. Sobald eine Headerdatei eine echte Funktion enthält, wird es mit dem mehrfachen Inkludieren schwierig. Dann nämlich würde der Compiler die entsprechende Funktion auch in mehreren Objektdateien unterbringen, was später zu einem Linkerfehler führen würde.

Die in den Objektdateien befindlichen Definitionen haben im Allgemeinen Abhängigkeiten untereinander. So übernimmt die Methode einer Klasse eventuell Parameter eines Typs, der in einer anderen Headerdatei deklariert ist, oder eine Klasse hat Attribute eines anderen Klassentyps. Diese Abhängigkeiten führen dazu, dass viele Headerdateien weitere Headerdateien inkludieren, die wiederum andere aufnehmen. Dabei werden häufig bestimmte Headerdateien mehrfach inkludiert. Damit nun keine Mehrfachdefinitionen vorgenommen werden, müssen die betroffenen Headerdateien gegen ein mehrfaches Compilieren gesichert werden:

Listing 6.1. Sicherung gegen Mehrfachinklusion

```
// Die Datei container.h
#ifndef __CONTAINER_H
```

```
#define __CONTAINER_H

class Liste
{
  // ...
};

class Vektor
{
  // ...
};
#endif // __CONTAINER_H
```

Diese Absicherung wird mit einem Präprozessordefine durchgeführt, das beim ersten Durchlauf definiert wird, aber bei jedem Durchlauf überprüft wird. Ist es schon definiert, wird der restliche Code der Headerdatei nicht mehr durchlaufen. Den Namen dieser Symbolischen Konstanten bildet man im Allgemeinen aus dem Dateinamen. Damit nicht zufällig der Name dieser Konstante mit irgendeinem Bezeichner im Code kollidiert, kann man dem Namen noch zwei Unterstriche voranstellen. Damit hat man die Wahrscheinlichkeit einer Namensgleichheit mit einem anderen Element im Code sehr stark verringert. Sollte die Verwendung des Dateinamens in einem Projekt immer noch nicht eindeutig sein, kann man noch den Pfad auf die Datei oder einen Teilpfad in den Namen der Symbolischen Konstanten aufnehmen.

Damit ist das mehrfache Deklarieren von Typen ausgeschlossen. Die Abhängigkeiten zwischen den Headerdateien führen aber immer noch dazu, dass auch andere Headerdateien inkludiert werden müssen, die in dem aktuell compilierten Modul gar nicht direkt gebraucht werden. Das führt zu einem Anstieg der Compilierzeiten. Eigentlich sind es immer die Abhängigkeiten zwischen den Headerdateien, die zu großen Compilierzeiten führen. In der Organisation eines Projektes ist die Art und Weise, wie man mit den Headerdateien umgeht, deshalb sehr wichtig. Dafür gibt es zwei wesentliche Stellschrauben, die in einem Projekt steuerbar sind.

Zum einen ist es die Anzahl der definierten Elemente in einer Headerdatei, die für ihre Abhängigkeit zu weiteren Headerdateien verantwortlich ist. Als Faustregel kann also gesagt werden, dass eine Headerdatei, je mehr Klassen sich in ihr befinden, auch mehr andere inkludieren muss. Außerdem wird sie auch selbst öfters inkludiert werden müssen, wenn sie mehr als eine Klasse enthält. Das heißt nun nicht, dass immer nur eine Klasse in einer Headerdatei aufgenommen werden sollte. Das hängt immer von der Art des Codes und der Größe des Projekts ab.

Zum anderen gibt es die Möglichkeit, eine Inklusionsreihenfolge von Projektheaderdateien zu definieren und eben nicht alle Headerdateien durch direkte Inkludierung voneinander abhängig zu machen. Also nicht:

Listing 6.2. Die Alles-Überall-Inklusion

```
// Header DB.h
class DBZugriff
{
  // ...
};
// Ende DB.h

// Header DBDialog.h
#include "DB.h"
class DBDialog
{
  DBZugriff *zugriff;
  // ...
};
// Ende DBDialog.h

// Applikation.cpp
#include "DBDialog.h"

int OpenDBDialog()
{
  DBDialog dlg;
  // ...
}
// Ende Applikation.cpp
```

sondern

Listing 6.3. Einhaltung einer Inklusionsreihenfolge

```
// Header DB.h
class DBZugriff
{
  // ...
};
// Ende DB.h

// Header DBDialog.h
class DBDialog
{
  DBZugriff *zugriff;
  // ...
};
// Ende DBDialog.h
```

```
// Applikation.cpp
#include "DB.h"       // Diese Datei muss vor
#include "DBDialog.h" // jener inkludiert werden!

int OpenDBDialog()
{
  DBDialog dlg;
  // ...
}
// Ende Applikation.cpp
```

Dieses Vorgehen reduziert zwar die Compilezeiten drastisch, erfordert aber mehr Wissen über die Abhängigkeiten der Headerdateien in einem Projekt. Vergisst man eine wichtige Headerdatei einzubinden, tritt ein Compilefehler in einer anderen Headerdatei auf. Die Fehlermeldung wird nur etwas darüber aussagen, welches Element nicht definiert ist. Für manche Entwickler kann das eine alptraumhafte Situation bedeuten – vor allem dann, wenn sie nicht an die Informationen kommen, welche Headerdateien sie vor welchen anderen einbinden müssen. Das macht dieses „flache Inklusionsmodell" für Headerdateien von Bibliotheken und Frameworks ungeeignet. Schließlich soll der Benutzer einer Bibliothek möglichst leicht mit ihr umgehen können und nicht zu viel Spezialwissen benötigen.

Insbesondere für Headerdateien in Bibliotheken und Frameworks gibt es eine weitere Methode, die Compilezeiten drastisch zu reduzieren: Abhängigkeiten zwischen Klassentypen kommen auch dadurch zustande, dass innerhalb einer Klasse eine Referenz oder ein Zeiger einer anderen Klasse als Attribut auftaucht. Möglicherweise übernimmt eine Klassenmethode auch einen Zeiger oder eine Referenz auf eine andere Klasse. C++ verlangt zur Deklaration eines Zeigers nicht die vollständige Typeninformation des zugrunde liegenden Klassen- oder Strukturtyps. Erst wenn auf die Referenz oder den Zeiger zugegriffen wird, muss die Typeninformation dem Compiler zur Verfügung stehen. Um Zeiger oder Referenzen eines bestimmten Typs zu deklarieren, ohne dessen volle Typinformation zu kennen, stellt C++ das Mittel der Vorwärtsdeklaration zur Verfügung. In einer Vorwärtsdeklaration deklariert man einfach den neuen Typ mit `class` oder `struct`, ohne einen Rumpf der Klasse anzugeben. Stattdessen schließt man die Deklaration des Typs nach dessen Bezeichner einfach ab. Also:

```
class Component;
struct ViewAttr;
```

Diese Zeilen deklarieren eine Klasse `Component` und eine Struktur `ViewAttr` vor, ohne deren Rumpf zu definieren. Damit ist dem Compiler grundsätzlich bekannt, dass es Typen mit den deklarierten Namen gibt, und er erlaubt die Definition von Zeigern darauf. Diese Zeiger kann der Compiler dazu verwen-

den, die Größe eines Typenlayouts zu berechnen, das solche Zeiger eventuell als Attribute enthält. Er kann auch die Größe des Stackbereichs berechnen, der von Übergabeparametern einer Methode benötigt wird. Zeiger haben an sich ja immer die gleichen Größe auf einem System, egal auf welchen Typ sie zeigen. Wenn allerdings in einer Implementierung auf einen solchen Zeiger zugegriffen wird, verlangt der Compiler die Typeninformation. Deshalb ist bei diesem Vorgehen auch die Definition von Inlinemethoden nicht möglich, wenn darin auf die vordeklarierten Typen zugegriffen werden muss.

Listing 6.4. Die Deklaration der Klasse `DBZugriff`

```
// Header DB.h
class DBZugriff
{
public:
  void init();
  // ...
};
// Ende DB.h
```

Die Klasse `DBDialog` enthält einen Zeiger auf `DBZugriff`. Wenn in der Headerdatei kein Zugriff über diesen Zeiger implementiert wurde, ist nicht die vollständige Typeninformation von `DBZugriff` notwendig. Es reicht eine Vorwärtsdeklaration, was die Abhängigkeiten der Headerdateien untereinander verringert und Compilezeit spart.

Listing 6.5. Vorwärtsdeklaration und Verwendung der Klasse `DBZugriff`

```
// Header DBDialog.h

class DBZugriff; // Vorwärtsdeklaration

class DBDialog
{
  DBZugriff *zugriff; // Hier wird keine vollständige
                      // Typeninformation gebraucht.

  // Die nachfolgende, auskommentierte Methode könnte
  // nicht übersetzt werden, da sie die Typeninformation
  // der Klasse DBZugriff benötigte.
  // void open() { zugriff->init(); }

  void open(); // Nur die Deklaration!
  // ...
};
// Ende DBDialog.h
```

In der Implementierungsdatei `DBDialog.cpp` muss die Headerdatei `DB.h` mit der vollständigen Typeninformation von `DBZugriff` inkludiert werden.

Listing 6.6. Einbindung der Typeninformation erst bei Zugriff auf Klassenelementen von `DBZugriff`

```
// DBDialog.cpp
#include "DB.h"
#include "DBDialog.h" // Hier wird die vollständige
                      // Typeninformation aufgenommen

void DBDialog::open()
{
  // Hier wird die Typeninformation gebraucht
  zugriff->init();
}
// Ende DBDialog.cpp
```

Man gewinnt durch dieses Vorgehen eine geringere Abhängigkeit der Headerdateien. Für die vordeklarierten Typen müssen keine extra Headerdateien mehr inkludiert werden. Damit sinkt die Compilezeit des Projekts und zirkuläre Beziehungen zwischen Klassen[2] werden realisierbar. Natürlich kann die vorgeschlagene Vorgehensweise auch in einem Projekt einer einfachen Anwendungsentwicklung durchgeführt werden. Im Allgemeinen zeichnet sich aber der Anwendungscode durch eine stärkere Kopplung der Dateien und Typen untereinander aus, da diese ja in Entwicklung sind. In einem Bibliotheks- oder Frameworkprojekt muss die Entkopplung der Module zu einem der wesentlichen Themen gemacht werden, sonst ist der Code später schlecht einsetzbar. Dort wird diese Technik eingesetzt, um Entkopplung zu erreichen.

6.1.2 Namensräume

Seit der ANSI/ISO-Spezifikation sind die Namensräume wichtiger Bestandteil aller C++-Entwicklungssysteme. Mit ihnen löst man das Problem, dass Typen gleichen Namens nicht gemeinsam in einem Projekt verwendet werden können, wenn man sie nicht durch eine zusätzliche Technik voneinander trennt. Diese Technik sind eben die Namensräume. Bis Mitte der 90er Jahre gab es sie nicht in C++ und es konnte das beschriebene Problem auftreten, dass Typen gleichen Namens miteinander in Konflikt gerieten. Problematisch war das überall dort, wo der Entwickler keinen Zugriff auf den Code hatte, wie

[2] Unter einer zirkulären Beziehung wird eine Beziehung verstanden, in der eine Klasse `A` Zugriff auf die Typeninformation einer Klasse `B` hat und andersherum. Es gibt bestimmte Situationen, in denen solche Typbeziehungen unumgänglich sind, wie zum Beispiel bei der Implementierung funktional sehr eng miteinander verbundener Klassen.

z. B. in Bibliotheken und Frameworks von Fremdherstellern. Diese können nun grundsätzlich durch Namensräume getrennt werden. Die gesamte Standardbibliothek vom C++ befindet sich beispielsweise im Namensraum `std`. Framework- und Bibliothekshersteller betten ihre Produkte in Namensräume ein, die meistens nach der Firma oder nach dem Produkt benannt sind.

Die Syntax

Man trennt also Code durch Namesräume. Dazu gibt es das Schlüsselwort `namespace`. Mit `namespace` und dem nachfolgenden Block, der mit geschweiften Klammern eingeschlossen wird, trennt man Code vom so genannten globalen Namensraum und von benachbarten Namensräumen. Der Namensraum erweitert den Typbezeichner und wird durch den Compiler beim Namemangling verwendet. Das heißt, es findet sich der Namensraum auch in einer verarbeiteten Form in den Objektdateien wieder. Das Namensraumkonzept findet sich auch in anderen Programmiersprachen wieder. So ist es beispielsweise mit dem Paketkonzept in Java vergleichbar. Es ist ein Strukturierungskonzept des Projekts und nicht des zu entwickelnden Systems. Durch die Namensräume erfährt das Modulkonzept eine Erweiterung. Aber nun zur Syntax der Namensräume:

Einen Namensraum definiert man folgendermaßen:

Listing 6.7. Syntax der Namensräume

```
namespace myLib
{
  class X { /* ... */ };
  struct Y { /* ... */ };
  typedef /* ... */ Z;
}
```

Da Bibliotheken und Frameworks aus Header- und CPP-Quellcodedateien bestehen, müssen die Namensräume in beiden Dateiarten definiert sein. Außerdem sind ja mehrere Dateien im Spiel, was ein ganz bestimmtes Verhalten von Namensräumen notwendig macht: Namensräume sind erweiterbar!

Listing 6.8. Erweiterbarkeit von Namensräumen

```
namespace mylib
{
  class A
  {
  public:
    void f();
  };
}
```

```
  // ...

namespace myLib
{
  void A::f()
  {
  }
}
```

Jede Headerdatei muss separat verwendet werden können. Deshalb steht darin eine Namensraumdefinition, die in weiteren Headerdateien erweitert werden kann. Dieses Prinzip liegt auch den Headerdateien der C++-Standardbibliothek zugrunde. Andernfalls könnten die verschiedenen Typen nicht in unterschiedlichen Headerdateien liegen.

Listing 6.9. Namensräume über mehrere Headerdateien

```
// A.h
namespace myLib
{
  class A { /* ... */ };
}

// B.h
namespace myLib
{
  class B { /* ... */ };
}
```

Eine weitere Eigenschaft von Namensräumen ist, dass sie verschachtelbar sind.

Listing 6.10. Verschachtelte Namensräume

```
namespace myLib
{
  namespace eventPart
  {
  // ...
  }
  namespace guiPart
  {
  // ...
  }
}
```

Wer Java kennt, dem wird hier die Verwandschaft zu den Paketen auffallen. Auch die Pakete sind verschachtelbar, bilden ihre Struktur aber, im Gegensatz zu C++, im Filesystem ab.

Um Bezeichner aus einem Namensraum zu verwenden, muss man nur den Namen des Namensraums mit dem Scope-Operator voranstellen. So enthält der Namensraum `std` einen Typ `string`. Wenn man auf diesen Typ zugreifen möchte, bezeichnet man ihn mit `std::string`. Alternativ kann man den Typ in einem Scope bekannt machen, indem man das Schlüsselwort `using` verwendet.

Die `using`-Direktive

Im nachfolgenden Listing wird gezeigt, wie man ein Element aus einem fremden Namensraum in den aktuellen Namensraum oder den aktuellen lokalen Bezeichnerraum einblendet.

Listing 6.11. Anwendung der `using`-Direktive auf ein Einzelelement

```
#include <string>

void f()
{
  using std::string;

  string s = "Text";

  // ...
}
```

Mit `using namespace` kann auch ein gesamter Namensraum eingeblendet werden.

Listing 6.12. Anwendung der `using`-Direktive auf einen Namensraum

```
#include <string>
#include <iostream>

void f()
{
  using namespace std;

  string s = "Text";
  cout << s << endl;

  // ...
}
```

Die beiden zuletzt beschriebenen Verfahren können auch innerhalb von Namensräumen verwendet werden.

Listing 6.13. Integration eines Elementes aus einem fremden Namensraum

```
namespace myPrj
{
  using std::string;
}
```

oder

Listing 6.14. Integration eines ganzen Namensraums

```
namespace myPrj
{
  using namespace std;
}
```

Weitere syntaktische Möglichkeiten der Namensräume

Zunächst soll hier noch die Aliasdefinition gezeigt werden.

Listing 6.15. Die Aliasdefinition

```
namespace X = myPrj;
```

Damit steht X für den Namensraum myPrj. Diese syntaktische Möglichkeit wird bis jetzt sehr selten genutzt und sie wurde hier vor allem der Vollständigkeit halber vorgestellt.

Die Erweiterung von Namensräumen um Implementierungen von Methoden kann auch dadurch erfolgen, dass bei der Definition eines Elements der Namensraumbezeichner direkt angegeben wird. Es ist nicht notwendig, einen Bezeichner, der schon innerhalb eines Namensraumes liegt, mit einem `namespace`-Block zu umgeben.

Listing 6.16. Erweiterung des Namens bei Elementdefinition

```
namespace myPrj
{
  class A
  {
  public:
    void f();
  };
}

// ...
```

```
void myPrj::A::f()
{
  // ...
}
```

Die direkte Angabe des Namensraums ermöglicht eine Überprüfung des Namensraums und des Bezeichners durch den Compiler. Noch nicht deklarierte Elemente werden durch den Compiler angemahnt, ungültige Namensräume ebenfalls.

Der Namensraum `std` in der ANSI/ISO C++ - Standardbibliothek

Eine für den Programmierer sofort sichtbare Veränderung des ANSI/ISO-Standards gegenüber den älteren AT&T-Standards ist das Fehlen der Endung `.h` an den Standard-Headerdateien. Aus `iostream.h` wurde also `iostream` und aus `string.h` wurde `string`. Diese neuen Headerdateien definieren die Bibliothekselemente innerhalb des Namensraumes `std`. Wenn also eine Übersetzung eines AT&T-konformen Codes innerhalb einer ANSI/ISO-Entwickungsumgebung durchgeführt werden soll, müssen die Includedateien angepasst und der Namensraum `std` eingeführt werden. Alter Code könnte folgendermaßen aussehen:

Listing 6.17. Programm im AT&T-Stil

```
#include <iostream.h>

// ...

int main()
{
  cout << "Eine Applikation im AT&T-Stil" << endl;
  // ...
  return 0;
}
```

Und so könnte man ihn auf den ANSI/ISO C++-Standard anpassen:

Listing 6.18. Programm im ISO-C++-Stil

```
#include <iostream> // Neue Headerdatei
using namespace std; // Nur zur Anpassung alten Codes

// ...

int main()
{
  cout << "Eine angepasste Applikation"
```

```
      << "im ANSI/ISO-Stil" << endl;
  // ...
  return 0;
}
```

ANSI/ISO-konformer Code sollte von vornherein Namensräume verwenden und bestehende Namensräume vorzugsweise lokal einführen. Also nicht:

Listing 6.19. Globale Öffnung des Namensraums `std`

```
#include <iostream>
using namespace std;

int main()
{
  cout << "Hello world!" << endl;

  return 0;
}
```

sondern

Listing 6.20. Lokale Öffnung des Namensraums `std`

```
#include <iostream>

int main()
{
  using namespace std;

  cout << "Hello world!" << endl;

  return 0;
}
```

Öffnet man Namensräume global, so schafft man eine Situation, in der man nur sehr schwer die Konfliktsituationen erkennt. Das lokale Öffnen der benötigten Namensräume vermeidet Konfliktsituationen und trainiert darüber hinaus das Bewusstsein für diese.

Zu beachten ist auch, dass selbst die alten C-Bibliothekselemente im Namensraum `std` zu finden sind. Wenn man sie verwenden möchte, muss man allerdings die spezielle Namensgebung der Headerdateien berücksichtigen. Aus der Datei `stdio.h` wurde `cstdio`, aus `stdlib.h` wurde `cstdlib`, usw. Es verschwindet also die Endung, und an den Anfang des Dateinamens wurde ein „c“ angehängt, um zu verdeutlichen, dass es sich um C-Bibliothekselemente handelt.

Der Grund für die Einführung anderer Namen ist in der möglichen zukünftigen Verwendung von C++ zu suchen. C++ ist eine sehr hardwarenahe

Sprache. Um diesen Vorteil gegenüber anderen, moderneren Sprachen wie Java oder C# ausnutzen zu können musste C++ für die Anwendungsbereiche fit gemacht werden, wo es auf exakt diese Eigenschaft ankommt. Das ist der Bereich der systemnahen Programmierung, der Treiberentwicklung und der Entwicklung für Embedded-Systeme. Gerade der letzte Bereich findet häufig auf minimalistischen Systemen statt; auf Systemen also, die eventuell über kein sehr leistungsfähiges Filesystem verfügen, die möglicherweise auch keine Dateiendungen kennen. Damit C++ überall in gleicher Weise einsetzbar wird, hat man für die Standardheaderdateien auf die Verwendung von Dateiendungen verzichtet.

Die Portierung alten Codes

Alter C++-Code, der noch nach einem AT&T-Standard geschrieben wurde, ist in einer ANSI/ISO C++-Entwicklungsumgebung nicht mehr unbedingt übersetzbar. Viele Compiler- und Bibliothekshersteller liefern Headerdateien mit, die die Bibliothekselemente auch im globalen Namespace definieren. Diese Headerdateien haben die Endung `.h` und enthalten neben der Includierung der Standardheaderdatei noch eine globale Direktive `using namespace std;`. Damit wird der Inhalt des Namensraumes `std` in den globalen Namensraum eingeblendet. Damit ist eine dem alten AT&T-Standard sehr ähnliche Voraussetzung gegeben.

Bei der Portierung alten C++-Codes vom AT&T-Standard auf den ANSI/ISO-Standard kann man nun genau so vorgehen, wie es beschrieben wurde. Man passt die Includedateien auf die Standardvarianten an und fügt in einer folgenden Zeile die Direktive `using namespace std;` ein. Viel weiter muss in den Code im Normalfall bezüglich der Includedateien nicht eingegriffen werden. Etwas mehr Schwierigkeiten macht unter Umständen die Anpassung des alten Codes an die neuen Eigenschaften des `new`-Operators.

Fazit

Hier wurde die Syntax und die Verwendung von Namensräumen in kleinen und übersichtlichen Codebeispielen gezeigt. Das muss nicht dazu verleiten, die Namensräume zur Kleinstrukturierung von Software einzusetzen. Man kann hervorragend ganze Bibliotheken und Frameworks voneinander trennen. Es existiert eine Diskussion darüber, ob Namensräume auch für Kleinstrukturen in Software eingesetzt werden sollten oder nicht. Auf jeden Fall stellen sie eine Strukturierung im Sinne von Domänenmodularisierung dar.

Allerdings sollte jeder nicht lokale Bezeichner in einen Namensraum aufgenommen werden. Die Namensräume sind dafür gedacht, den globalen Namensraum zu leeren. Bei der Implementierung von Methoden bietet sich die `namespace::member`-Schreibweise an. Der Compiler kann überprüfen, ob der Namensraum existiert und ob ein Element des entsprechenden Namens

bereits deklariert ist. Ausserdem verdeutlicht diese Schreibweise die Zugehörigkeit der definierten Elemente für den Programmierer besser als die Schreibweise mit dem umfassenden Block.

6.1.3 Das argumentenabhängige Lookup - Verfahren

An dieser Stelle soll ein einfaches Programm betrachtet werden, um ein Standardverfahren zu verdeutlichen. Sehen Sie sich also das nachfolgende Programm einmal sehr genau an und stellen Sie sich die folgenden Fragen:

1. In welcher Headerdatei ist der Operator << definiert?
2. Gehört der Operator zu einer Klasse oder ist er global?
3. In welchem Namensraum muss er definiert sein?
4. Wie wird er durch den Compiler gefunden?

Listing 6.21. Schritt 1: Ein sehr einfaches Programm

```
#include <iostream>
#include <string>

int main()
{
  std::string str = "Na sowas!";

  std::cout << str;

  return 0;
}
```

Strukturell scheint das Programm in Listing 6.21 sehr einfach zu sein. Die Herkunft des Typs `std::string` und des Objekts `std::cout` dürfte klar sein. Dazu wurden die Includedateien `<iostream>` und `<string>` aufgenommen. In einer der beiden Dateien muss nun auch der Operator << deklariert sein, der den linken Operanden `ostream` und den rechten `string` entgegennimmt. Wenn der Operator zu einer Klasse gehören würde, müsste er zu `ostream` gehören. In diesem Fall müsste er in `<iostream>` deklariert sein. Dort müsste dann auch der Typ `string` bekannt sein, was eine Abhängigkeit zwischen `ostream` und `string` nach sich ziehen würde. Das Includieren von `<string>` würde sich damit erübrigen. Wenn Sie jedoch den Test machen und versuchen das Programm ohne `<string>` zu compilieren, werden Sie feststellen, dass es ohne diese Includedatei nicht geht. Daraus folgt, dass der gesuchte Operator nicht in `ostream` deklariert sein kann. Er muss dort deklariert sein, wo `string` bekannt ist, zumal es sich bei `string` nicht um eine einfache Klasse handelt, die vordeklariert werden kann, sondern um eine Templateinstanziierung. Der Operator steht also in `<string>` und muss deshalb global sein. Der folgende Code muss also übersetzbar sein:

Listing 6.22. Schritt 2: Woher kommt der Ausgabeoperator?

```
#include <iostream>
#include <string>

int main()
{
  std::string str = "Na sowas!";

  operator<<(std::cout,str);

  return 0;
}
```

Hierbei handelt es sich um eine reine Umformulierung dessen, was im Listing 6.21 steht. Der Compiler macht diese Umformung von sich aus. In Listing 6.22 sind wir gewissermaßen dem Compiler einen Schritt vorausgegangen. Es fällt dabei auf, dass kein Namensraumbezeichner am Operator steht. Der Operator wird trotzdem gefunden. Wenn die Aussage wahr sein soll, dass alle Elemente der Standardbibliothek im Namensraum `std` stehen, dann müsste der Code eigentlich aussehen wie in Listing 6.23:

Listing 6.23. Schritt 3: Der Operator « kommt aus `std`

```
#include <iostream>
#include <string>

int main()
{
  std::string str = "Na sowas!";

  std::operator<<(std::cout,str);

  return 0;
}
```

Wenn wir den Code testen, stellen wir fest, dass auch er übersetzbar ist. Es gibt also den verwendeten Operator im Namensraum `std`. Hinterfragt man den Grund für die Lauffähigkeit von Listing 6.22 und Listing 6.23, so kann man sich die Frage nach der Herkunft des Operators stellen. Handelt es sich bei dem Operator ohne die Namensraumangabe um denselben Operator, oder ist durch eine separate globale Definition der Zugriff auf den Operator im Namensraum `std` gesichert worden?

Wenn es sich nicht um eine separate globale Definition handelt, dann muss der Compiler ein Suchverfahren haben, das das Auffinden des Operators im Namensraum auch ohne dessen Angabe ermöglicht. Dieses Verfahren nennt

man das argumentenabhänge Lookup-Verfahren[3]. Es lässt sich auch mit ganz normalen Funktionen durchführen. Der folgende Code in Listing 6.24 ist beispielsweise gültig und wird übersetzt:

Listing 6.24. Logische Folge aus den Schritten 1 - 4

```
namespace test
{
  class X {};

  void f( X ) {}
}

int main()
{
  test::X x;

  f( x ); // test::f( test::X ) !

  return 0;
}
```

Der Code demonstriert auf einfache Weise das argumentenabhängige Lookup-Verfahren. Funktionen werden nach diesem Verfahren nicht einfach auf dem aktuellen Scope oder dem globalen Namensraum gesucht, sondern auch innerhalb der Namensräume der Funktionsparameter. Andernfalls wäre das einfache Programm aus dem ersten Listing dieses Abschnitts nicht übersetzbar.

Es muss allerdings noch hinzugefügt werden, dass die aufgeführten Beispiele nur mit einem zumindest näherungsweise standardkonformen Compiler nachvollzogen werden können. Ältere Compiler implementieren das argumentenabhängige Lookup-Verfahren nicht oder nur teilweise.

Das beschriebene Lookup-Verfahren wird im Englischen als Argument Dependent Lookup oder kurz als ADL bezeichnet.

6.1.4 Einige Anmerkungen zum Linker

In C++ existieren einige Sprachmerkmale, die eine Erweiterung der Zusammenarbeit zwischen Compiler und Linker benötigen. Die klassische Zusammenarbeit zwischen diesen Werkzeugen in der C-Programmierung sieht so aus, dass der Compiler Objektdateien erzeugt, die durch den Linker zu einer ausführbaren Datei zusammengebunden werden. Die Objektdateien wer-

[3] Es hat auch den Namen „Koenig-Lookup“ von seinem Erfinder Andrew Koenig.

den unabhängig voneinander durch den Compiler erzeugt. In ihnen finden sich benannte Symbole mit ihren Implementierungen in Maschinensprache (Funktionen und globale Variaben). Wenn der Linker die Objektdateien bindet, dürfen Bezeichner nicht doppelt vorkommen. Bei der Zuordnung der Symbole verwendet der Linker ausschließlich diese Bezeichner und nicht etwa irgendwelche Typeninformationen von Variablen und Parameterlisten. Die Typeninformationen gehen im Compilerlauf vollständig verloren. Damit keine doppelten Symbole in den Objektdateien vorkommen, verbietet C die Verwendung gleicher Bezeichner für mehr als ein bestimmtes Element. Wenn aus irgendeinem Grund doch zwei gleiche Symbolnamen in den zu linkenden Dateien gefunden werden, gibt der Linker eine Fehlermeldung aus.

C++ erlaubt die Verwendung des gleichen Bezeichners für Funktionen und Methoden mit unterschiedlichen Parameterlisten. Um nicht den Bezeichner mehrfach in die Objektdateien aufzunehmen erzeugt der Compiler einen Symbolnamen und verwendet dazu auch die Typen der Parameterliste. Das Verfahren ist nicht standardisiert und deshalb auch nicht kompatibel zwischen Compilern. Es können keine Objektdateien gemeinsam gebunden werden, die von unterschiedlichen Compilern stammen[4]. Das Verfahren zur Bildung des Symbolnamens aus Namespace, Klassenzugehörigkeit, Namen und Parameterliste nennt man Namemangling. Damit erreicht man die Eindeutigkeit der Symbolnamen für den Linker. Andere Merkmale von C++ lassen sich nicht mehr in garantiert eindeutige Symbolnamen abbilden. Da ist zum Beispiel die Möglichkeit der `inline`-Deklaration von Methoden. Um solche Methoden in mehreren Modulen verwenden zu können, müssen diese in Headerdateien definiert werden. Die `inline`-Deklaration ist aber nur eine Empfehlung an den Compiler, die dieser nicht unbedingt umsetzten muss. Setzt der Compiler die `inline`-Deklaration nicht um und wird die Headerdatei in mehreren Modulen inkludiert, dann werden mehrere gleiche Elemente mit gleichem Symbolnamen in die Objektdateien aufgenommen. Das ist beispielsweise bei virtuellen Funktionen[5] der Fall.

Gleiches geschieht eventuell bei der Instanziierung von Templates[6]. Wenn Templates Implementierungen enthalten und in mehreren Modulen mit den gleichen Templateparametern instanziiert werden, erzeugt der Compiler auch identische Elemente. Ähnliches geschieht bei der Deklaration von Klassen, die virtuelle Methoden enthalten: die Klassen brauchen einen Zeiger auf die V-Tabelle und natürlich die V-Tabelle selbst. Da Klassen gewöhnlich in den Headerdateien deklariert werden und diese wiederum in mehreren Modulen inkludiert werden, erzeugt der Compiler für mehrere Module die gleichen Symbole. Zur Verdeutlichung des Problems sollen die folgenden Beispiele beitragen:

[4] In ANSI-C ist es möglich.
[5] Virtuelle Funktionen werden im Abschnitt 2.2.30 beschrieben.
[6] Mehr zu den Templates steht im Abschnitt 4.1 ab Seite 222.

Listing 6.25. Das Template `min()`

```
// Headerdatei MIN.H
template <class T>
const T& min( const T& t1, const T& t2 )
{
  return ( t1 < t2 ) ? t1 : t2;
}
```

Listing 6.26. Die Benutzung eines Templates in mehreren Modulen

```
// Modul F1.CPP
#include "MIN.H"
int f1( int a, int b )
{
  // ...
  int x = min( a, b );
  // ...
}

// Modul F2.CPP
#include "MIN.H"
int f2( int a, int b )
{
  // ...
  int x = min( a, b );
  // ...
}
```

In den Modulen F1 und F2 wird das gleiche Funktions-Template instanziiert:

```
min<int,int>(const int&,const int&)
```

Da die Module unabhängig voneinander compiliert werden, steht der Compiler jedesmal vor der Notwendigkeit, das instanziierte Template als echte Funktion in die Objektdatei zu schreiben. Erst beim Linken stoßen die gleichen Symbole aufeinander.

Listing 6.27. Inklusion einer Klassendeklaration mit V-Tabelle in mehreren Modulen

```
// Headerdatei BASIS.H
class Basis
{
public:
  virtual ~Basis();
};
```

```
// Modul BASIS.CPP
#include "BASIS.H"
Basis::~Basis()
{
}

// Modul KONKRET.CPP
#include "BASIS.H"

class Konkret : public Basis
{
};

int main()
{
  Konkret k;
  return 0;
}
```

Diese drei kleinen Listings zeigen das nächste Problem. Eine Klasse mit virtuellen Methoden hat eine V-Tabelle, in der die Methoden verlinkt sind. Die Objekte der Klasse verweisen auf die V-Tabelle. Im Zusammenhang mit der V-Tabelle steht etwas Funktionalität, die der Compiler zur Klasse hinzugeneriert. Die V-Tabelle selbst ist etwas, das als statisches Element einer Klasse aufgefasst werden kann. Im C++-Quelltext ist dies nicht direkt sichtbar, in den Objektdateien aber schon. In welchem Modul muss der Compiler diese statischen Elemente anlegen? Er weiß ja wieder nichts von den jeweils anderen Modulen. Entweder er legt diese Elemente in jedem Modul an, das die Deklaration der Klasse – in unserem Beispiel `Basis` – enthält, oder er verfolgt eine andere Strategie. Manche Compiler legen die zur V-Tabelle gehörenden statischen Elemente in dem Modul an, in dem die erste virtuelle Methode der Klasse definiert ist. Im letzteren Fall muss der Linker keine besondere Arbeit leisten. Der erste Fall aber nötigt dem Linker wieder die Arbeit auf, unnötigen Code aus dem Programm zu entfernen.

Die beschriebenen Situationen führen bis heute zu Problemen in der Softwareentwicklung, wenn der Linker nicht einige Fähigkeiten hat, die es ihm erlauben mehrfache Symbole zu entfernen. Manche Entwicklungssysteme optimieren durch einen Prelinker oder eine bestimmte Compilephase die Objektdateien in einem Projekt. Die Compilephase hat den Vorteil, dass der Linker nicht mehr entfernen muss, was der Compiler zeitaufwändig erarbeitet hat. Es müssen dafür aber Informationen zwischen den Compileprozessen der Module ausgetauscht werden, was meistens über zusätzliche Dateien geschieht.

Grundsätzlich sollte aber ein moderner Linker für C++ die Fähigkeit zur Optimierung haben. Überflüssige Elemente sollten aus den Objektdateien herausgelöscht werden können. Leider ist das noch nicht immer der Fall. Der Linker arbeitet eher im Hintergrund und macht sich nur selten bemerkbar, während der Compiler dem Entwickler eine schnellere und viel differenziertere Antwort auf Fehler gibt. Das führte möglicherweise dazu, dass an den Compilern mehr gearbeitet wurde als an den Linkern. Möglicherweise spielte auch die Überlegung mit, dass etwas größere ausführbare Dateien in den meisten Fällen kein Problem darstellen[7]. In großen Projekten sind solche Linker allerdings ein echtes Problem. Sie führen dazu, dass manche C++-Techniken und Sprachmerkmale gar nicht oder nur in einer sehr eingeschränkten Weise eingesetzt werden können. Diese Techniken müssen dann durch aufwändige Konstrukte ersetzt werden, die den Softwareentwurf verkomplizieren und auch fehleranfälliger machen. In umfangreichen Projekten wird deshalb auch oft ein spezieller Linker eingesetzt und nicht der, der standardmäßig im Entwicklungssystem enthalten ist.

6.1.5 Überladen der Operatoren `new` und `delete`

Die Operatoren `new` und `delete` sind nicht einfach Operatoren wie andere auch. Wenn man sie neu definiert, muss man schon einiges über die Arbeitsweise der Speicherverwaltung auf dem Zielsystem verstehen, sonst wird die Arbeit wahrscheinlich von weniger Erfolg gekrönt sein. Insbesondere in Software mit speziellen Anforderungen an den Umgang mit Speicher können solche Überladungen notwendig werden: in Embedded-Systemen, in Systemen mit physikalischer Speicherverwaltung oder einfach auf Systemen mit sehr knappem Speicher. Vielleicht ist es aber auch ein bestimmter Objekttyp, der so zahlreich instanziiert und wieder freigegeben werden soll, dass sich eine spezielle Verwaltung für ihn lohnt.

Wie andere Operatoren auch, können `new()` und `delete()` überladen werden. Dabei werden diese beiden Operatoren nicht als Objektmethoden definiert, sondern als statische Methoden in der Klasse, für die sie überladen werden. Es ist nicht möglich `new()` und `delete()` als globale Methode für einen benutzerdefinierten Typ zu überladen. Diese beiden Operatoren müssen immer Elemente einer Klasse sein.

[7] In der 16-Bit-Programmierung unter DOS, Windows 3.1 und Mac-Systemen war es ein sehr großes Problem, da man sehr auf die Größe der Programmdatei achtgeben musste. Heute hat man die Problematik in der Entwicklung von Embedded-Systemen.

new

Die Deklaration eines solchen benutzerdefinierten `new()`-Operators sieht folgendermaßen aus:

Listing 6.28. Überladener `new`-Operator

```
class X
{
public:
  void* operator new( size_t );
  // ...
};
```

Der Parameter des `new()`-Operators gibt die Größe des zu allokierenden Objekts an. Diese Information kann durch den Entwickler an die verwendete – oder selbst implementierte – Speicherverwaltung weitergereicht werden.

Der `new`-Operator kann auch mit weiteren Parametern deklariert werden. Im nachfolgenden Beispiel wird ein Funktionszeiger auf eine Fehlerbehandlungsroutine transportiert.

Listing 6.29. Überladener `new`-Operator mit Fehlerroutine

```
class X
{
public:
  void* operator new( size_t, new_handler );
  // ...
};
```

Der Datentyp `new_handler` ist in der Headerdatei `<new>` wie folgt definiert:

```
typedef void (*new_handler)();
```

Es ist der Typ eines Funktionszeigers auf Funktionen ohne Parameter und ohne Rückgabewert[8]. Das Problem an der Sache mit solchen `new()`-Operatoren ist, dass entsprechende `delete()`-Operatoren nicht in der Weise deklariert werden können, dass sie im Fall eines Stackabbaus automatisch aufgerufen werden. Das verursacht bei vielen Compilern eine Warnung.

Die beiden durch den C++-Standard definierten zweiparametrigen `new`-Operatoren sind die Placement- und die `nothrow`-Variante. Auch diese beiden können redefiniert werden.

delete

Der `delete()`-Operator bekommt als Parameter den Zeiger auf das freizugebende Objekt; genauer gesagt einen Zeiger auf den freizugebenden Speicher-

[8] Ein solcher Handler könnte im Fehlerfall eine bestimmte Exception werfen oder in irgendeiner Weise in die Speicherverwaltung eingreifen.

block, denn der Destruktor ist bereits aufgerufen, wenn der Programmlauf im Implementierungsblock des `delete()`-Operators ankommt.

Listing 6.30. Überladener `delete`-Operator

```
class X
{
public:
  void operator delete( void * );
  // ...
};
```

Wie schon angesprochen, ist es eine Besonderheit dieser beiden Operatoren, dass sie statische Elemente von Klassen sind. Die syntaktische Besonderheit dabei ist, das die statische Elementeigenschaft nicht deklariert werden muss. Sie kann aber deklariert werden, was die Art dieser beiden Klassenelemente besser sichtbar macht:

Listing 6.31. Explizite statische Deklaration von `new`- und `delete`-Operatoren

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

Der `delete()`-Operator hat noch eine Variante mit einem zweiten Parameter, der die Größe des freizugebenden Blocks anzeigt:

```
void operator delete( void *, size_t );
```

Dabei kann der `delete()`-Operator nicht überladen werden, indem beide Varianten für eine Klasse definiert werden. Hier gibt es nur den mit einem Parameter oder den mit den zweien.

Der Typ `size_t` stammt aus der Headerdatei `<cstddef>`, die zur Überladung der Operatoren `new` und `delete` immer inkludiert werden muss.

Bevor wir uns mit den möglichen Implementierungen dieser Operatoren befassen, sehen wir uns noch die Anwendung an. Bei der dynamischen Objektinstanziierung mittels `new` wird der überladene Operator verwendet, wenn er vorhanden ist:

```
X *p = new X();
```

Es gibt aber weiterhin die Möglichkeit, den alten globalen `new`-Operator zu verwenden, indem man diesen im globalen Namensraum sucht:

```
X *p = ::new X();
```

Die Varianten new[] und delete[]

Ebenso wie die normalen Varianten von `new` und `delete` können die Array-Varianten überladen werden. Der Parameter `size_t` des `new`-Operators gibt die Gesamtgröße des Speichers eines Arrays an. Es berechnet sich also aus der Anzahl der zu allokierenden Elemente und aus deren Größe: `n * sizeof(X)`. Für einen überladenen Operator `new` muss auch ein überladener Operator `delete []` zur Verfügung gestellt werden.

Listing 6.32. Deklarationspaare von `new`- und `delete`-Operatoren

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

Die Funktionsweise des Paars `new[]`-`delete[]` ist dem Paar für die Allokation von Einzelobjekten `new`-`delete` grundsätzlich vergleichbar; mit der einen Ausnahme, dass `new[]` mehrere Konstruktoren aufruft. Für jedes Objekt in einem Array muss schließlich ein Konstruktor aufgerufen werden. Entsprechend müssen bei der Freigabe der Objekte auch wieder Destruktoren aufgerufen werden. Deshalb muss `delete[]` verwendet werden. Dieser Operator ruft vor der eigentlichen Speicherbefreiung alle Destruktoren der in dem Array vorhandenen Objekte auf[9].

Regeln für das Überladen von new und delete

Sowohl in der eigenen Implementierung des `new`- wie auch des `delete`-Operators muss man die damit verbundenen Klassenmethoden – Konstruktor und Destruktor – nicht explizit aufrufen. Wenn die `new`-Implementierung aufgerufen wird, muss nur der Speicher zurückgeliefert werden, in dem das Objekt platziert wird. Der Compiler sorgt dafür, dass später der entsprechende Konstruktor aufgerufen wird. Für den `delete`-Operator wird der Aufruf des Destruktors ebenso automatisch durch den Compiler eingefügt. Der Destruktor wird allerdings vor dem Aufruf der `delete`-Implementierung ausgeführt. In diesem Bereich wird man durch den Compiler unterstützt. Es gibt jedoch andere Aspekte, um die man sich kümmern muss.

Zum einen muss man darauf gefasst sein, dass ein überladener `new`-Operator für die Instanziierung eines abgeleiteten Typs verwendet wird. Zum an-

[9] Würde man ein solches Array mit dem `delete`-Operator für Einzelobjekte löschen, würde nur der Destruktor des ersten Elements aufgerufen werden. Der Speicher würde womöglich ganz freigegeben werden, was aber von der jeweiligen Implementierung des Operators abhängt.

deren wird eine bestimmte Ausnahmebehandlung durch einen `new`-Operator erwartet.

Der folgende Code demonstriert, dass die Operatoren `new` und `delete` vererbt werden und daher mit unterschiedlichen Speicherblockgrößen rechnen müssen.

Listing 6.33. Demo zur Überladung von `new` und `delete`

```
#include <cstddef>
#include <new>
#include <iostream>

class X
{
public:
  static void* operator new ( std::size_t s );
  static void  operator delete ( void *, std::size_t );
};

class Y : public X
{
  public:
  char array[1000];
};

void* X::operator new ( std::size_t s )
{
  std::cout << s
            << " Bytes durch new angefordert."
            << std::endl;
  return ::operator new (s);
}

void  X::operator delete ( void *p, std::size_t s )
{
  ::operator delete( p );
  std::cout << s
            << " Bytes durch delete feigegeben."
            << std::endl;
}

int main()
{
  X *p1 = new X();
  Y *p2 = new Y();
```

```
  delete p1;
  delete p2;

  return 0;
}
```

Die Ausgabe unterscheidet sich bei verschiedenen Compilern nur unwesentlich. Gemäß dem Standard können Objekte niemals die Größe null Bytes haben. Sie haben in dem Fall, dass keine Attribute in der Klasse definiert wurden, die Größe 1. Manche Compiler erweitern diese Größe auf 4 oder 8 Bytes. Bei diesen Größen handelt es sich um Pro-forma-Werte, die der Standard verlangt, um Objekte mit null Bytes unmöglich zu machen. Da ein Klassenobjekt ohne Attribute auch nicht auf einen eigenen Speicherplatz angewiesen ist, kann diese Pro-forma-Größe bei einer Vererbung der Klasse wegoptimiert werden.

Die Ausgabe, die beispielsweise die Compilate der Compiler MS Visual C++ 7.1, GCC C+ 3.4.4 unter Windows erzeugen:

```
1 Bytes durch new angefordert.
1000 Bytes durch new angefordert.
1 Bytes durch delete feigegeben.
1000 Bytes durch delete feigegeben.
```

Der Borland C++ 5.5.1 erzeugt ein Compilat, das folgende Ausgabe zeitigt:

```
8 Bytes durch new angefordert.
1008 Bytes durch new angefordert.
8 Bytes durch delete freigegeben.
1008 Bytes durch delete freigegeben.
```

Man sieht also, dass man in einer `new`-Implementierung für eine beliebige Klasse daran denken muss, dass dieser Operator in einem Kontext aufgerufen wird, in dem er Speicher nicht nur für den Objekttyp zur Verfügung stellen muss, in dem er deklariert ist. Das ist beispielsweise dann wichtig, wenn man mit Hilfe überladener `new-delete`-Paare eine Objektpoolverwaltung schreiben möchte. Man kann nicht davon ausgehen, dass die angeforderten Speicherblöcke immer die gleiche Größe haben. Dafür muss man eine Abfrage einbauen, die etwa wie in Listing 6.34 aussehen könnte.

Listing 6.34. Operator `new` mit Sicherungsabfrage

```
void* X::operator new( std::size_t s )
{
  if( s != sizeof(X) )
    return ::operator new(s);

  return getMemoryFromXPool();
}
```

Dieser `new`-Operator nutzt eine Objektpoolverwaltung für die Klasse `X`. Wenn eine Speicheranforderung durch die Instanziierung einer abgeleiteten Klasse erfolgt, delegiert er diese an den Standard-`new`-Operator.

Ebenso muss sich der `delete`-Operator verhalten. Der zweite Parameter, der die Objektgröße angibt, kann für einen solchen Fall zur Unterscheidung der Freigabemethoden herangezogen werden:

Listing 6.35. Operator `delete` mit Sicherheitsabfrage

```
void  X::operator delete ( void *p, std::size_t s )
{
  if( s != sizeof(X) )
    ::operator delete( p );
  else
    giveMemoryBackToPool( p );
}
```

Die zweite angesprochene Problematik ist die, dass man – will man ein einigermaßen standardkonformes Verhalten des Operators `new` erreichen – ein ganz bestimmtes Verfahren einhalten sollte, das die Fehlermöglichkeiten und deren Behandlung abdeckt. Das erreicht man unter anderem dadurch, dass man bei Fehlern an die Standardimplementierung `::operator new` durchreicht, damit diese standardkonform reagieren kann[10]. Sicherlich ist das nicht in jedem Fall akzeptabel und man muss einen eigenen Standard-`new`-Operator schreiben. Wie das geht, steht in dem Buch „Effektiv C++ programmieren" von Scott Mayers [Meyers98] und soll an dieser Stelle kurz skizziert werden:

Listing 6.36. Schema des globalen `new`-Operators aus [Meyers98]

```
void* operator new(size_t size)
{
  // Eine 0-Byte Anforderung wird als
  // 1-Byte Anforderung behandelt.
  // Nach dem Standard ist eine Anforderung
  // von 0 Byte gültig.
  if( size == 0 ) size = 1;

  for(;;)
  {
    Versuch, Speicher zu allokieren;
    if( Versuch erfolgreich )
      return (Zeiger auf Speicher);

    // Die Allokation ist fehlgeschlagen.
    // Jetzt muss die Fehlerbehandlungsfunktion
    // ermittelt werden.
```

[10] Eventuell den Speicher vom Heap holen oder eine Exception werfen.

```
    new_handler globalHandler = set_new_handler(0);
    set_new_handler( globalHandler );

    if( globalHandler ) (*globalHandler)();
    else throw std::bad_alloc();
  }
}
```

Zum schematischen Beispiel in Listing 6.36: Eine Anforderung nach null Bytes ist gemäß dem Standard eine gültige Anforderung und muss bearbeitet werden. Damit man eine Adresse zurückliefern kann, erhöht man die Anfrage auf 1 Byte. Danach wird in eine Endlosschleife eingetreten, die bei erfolgreicher Allokation direkt verlassen wird. Konnte allerdings kein Speicher besorgt werden, kommt die Schleife eventuell zum Tragen. Es wird der gültige Handler bestimmt[11] und aufgerufen. Dieser Handler muss irgendwie Speicher beschaffen, denn danach wird die Allokation wiederholt. Schlägt auch diese fehl, wird erneut der Handler gerufen und so weiter. Wenn der Handler keinen Speicher besorgen kann, wirft er `bad_alloc`. Wenn in der Schleife des `new`-Operators auf einen Fehlerfall kein Handler besorgt werden kann, wird auch `bad_alloc` geworfen.

6.2 Persistenz und Serialisierung

In C++ besteht ein ganz grundsätzliches Problem, wenn es darum geht, Objekte über die Laufzeit eines Programms hinaus persistent zu machen, d. h. wenn Objekte „abgespeichert“ und während eines späteren Programmlaufs „eingelesen“ werden sollen. Dabei ist es egal, ob Objekte in einer Datenbank oder einfach als Datei abgespeichert werden sollen.

Das Problem besteht darin, dass es keine eigentliche Typeninformation gibt, die mit dem Objekt abgespeichert werden kann[12]. Außerdem gibt es kein standardisiertes Verfahren, wie das Einlesen vorgenommen werden kann. C++ hat ein sehr spezifisches Typensystem, das noch nicht einmal zwischen unterschiedlichen Compilern auf dem gleichen System identisch ist.

Nehmen wir also an, es sollen x Objekte in einen Datenstrom abgespeichert werden. Dabei muss jedes Objekt seine Typenidentifikation in den Strom schreiben und zusätzlich die Werte seiner Attribute. Wenn sich die Objekte in ihrem Typ unterscheiden, brauchen diese eine gemeinsame Basisklasse, die eine rein virtuelle Methode zum Speichern der Daten deklariert.

[11] Dabei gibt es leider keine Funktion `get_new_handler()` in der Standardbibliothek.

[12] In Java gibt es genau diese Typeninformation. Da das Typensystem in Java auf allen Virtual Machines gleich ist, können Objekte samt der Typeninformation gespeichert werden und durch eine beliebige VM wieder eingelesen werden.

Listing 6.37. Schematische Deklaration zur Persistenz

```
class PersistenzObjekt
{
public:
  virtuel ~PersistenzObjekt();

  virtual void schreibe( std::ostream & ) = 0;
};
```

Die konkrete Klasse kann dann von dieser Basisklasse erben und die entsprechende Methode implementieren.

Listing 6.38. Schematische Implementierung zur Persistenz

```
class Punkt : public PersistenzObjekt
{
public:
  // ...

  void schreibe( std::ostream & );

private:
  int x, y;
};

void Punkt::schreiben( std::ostream &os )
{
  os << "Typ: Punkt" << std::endl;
  os << x << std::endl;
  os << y << std::endl;
}
```

Beim Einlesen stellt sich nun die Frage, wer die abgespeicherte Typeninformation liest. Das Objekt selbst kann es nicht tun. Es existiert ja noch nicht. Es muss also eine von den abgespeicherten Typen unabhängige Instanz sein. *Das ist der eigentliche Problempunkt an der Serialisierung von Objekten. Es muss erstens eine Typeninformation generiert werden, um den Typ in den persistenten Daten zu kennzeichnen. Zweitens muss es ein Programmteil geben, um diese generierte Typeninformation zu lesen und den richtigen Objekttyp dafür zu instanziieren.* Das kann beispielsweise eine unabhängige Funktion sein. Oder das Objekt, das die Datenbank oder den Stream repräsentiert, könnte eine solche Methode zur Verfügung stellen. In einer solchen Funktion oder Methode müssen alle Typen bekannt gemacht werden, die potentiell in persistenter Form vorliegen können.

Listing 6.39. Einlesende Funktion

```
PersistenzObjekt* einlesen( std::istream &is )
{
  PersistenzObjekt *obj = NULL;
  std::string typestring;
  is >> typestring;
  if( /* überprüfe typestring auf Punkt */ )
  {
    int x, y;
    is >> x;
    is >> y;
    obj = new Punkt(x,y);
  }
  else if( /* Überprüfe typestring auf sonstwas */ )
  {
    // ...
    obj = ...
  }

  return obj;
}
```

Die Situation ist prinzipiell etwas unbefriedigend, da die Kopplung der Deserialisierungsfunktion zu den persistenten Datentypen sehr hoch ist. Es gibt da ein paar Techniken, um diese Kopplung in ihren Auswirkungen etwas abzumildern. Mann kann beispielsweise den persistenten Typen Konstruktoren geben, die extra für die Deserialisierung geschrieben werden. Ein solcher Konstruktor könnte beispielsweise einen Stream entgegennehmen und die eigenen Attribute selbst einlesen.

Listing 6.40. Konstruktor zum Einlesen eines Objekts

```
Punkt::Punkt( std::istream &is )
{
  is >> x;
  is >> y;
}
```

In vielen Bibliotheken und Frameworks sind die beschriebenen Elemente hinter Makrodefinitionen versteckt oder werden durch Generatoren hinzugefügt. Damit wird der Aufwand für die Serialisierung reduziert.

6.3 Systemprogrammierung mit C++

Auch wenn sich unter manchen Systemprogrammierern hartnäckig das Gerücht hält, dass C++ wegen des „Overheads", das die Objektorientierung mit sich bringe, nicht für die Systemprogrammierung geeignet sei und nur C die Sprache der Wahl sein könne, erfreut sich C++ gerade in diesem Bereich zunehmender Beliebtheit. Bei Licht betrachtet löst sich das Argument mit dem Overhead auch schnell in Wohlgefallen auf, denn schließlich hat man in C++ die Möglichkeit, jederzeit OO-Methoden mit den prozeduralen „alten/-grq/ C-Methoden zu verbinden und eine beliebige Mischform zu verwenden. Hinzu kommt, dass der C++-Compiler im Gegensatz zu seinem C-Pendant eine strengere Typenüberprüfung durchführt und schon aus diesem Grund mehr Informationen zur Compilezeit zur Verfügung hat, um Optimierungen durchzuführen. Ganz besonders kommt das beim Einsatz der Templatetechniken zum Tragen, denen C nichts entgegenzusetzen hat. Natürlich kommt es darauf an, die in der Systemprogrammierung verwendeten Techniken nach Effizienzkriterien zu wählen. Dabei ist unter anderem die STL ein starkes Argument zugunsten von C++. Des Weiteren ist die Möglichkeit zur Operator- und Funktionsüberladung eine Technik, die ohne Effizienzverlust eine bessere Strukturierung von Systemcode erlaubt. In manchen Fällen kann der C++-Compiler durch überladene Funktionen eine Entscheidung zur Compilezeit treffen, die in einem C-Code zur Laufzeit erfolgt wäre. In diesem Kapitel werden einige dieser C++-Techniken vorgestellt, die in systemnahen Code zum Einsatz kommen können und dort entweder eine bessere Strukturierung von Code ermöglichen oder sogar zur Effizienzverbesserung führen.

6.3.1 Einfaches Objektpooling

Bestimmte Bereiche der Systemprogrammierung lassen sich sehr gut mit objektorientierten Methoden bearbeiten. Da ist zum Beispiel das Problem der Speicherverwaltung. Auf bestimmten Systemen muss man versuchen, die Fragmentierung, die durch viele Allokationen und Deallokationen kleiner Objekte entstehen kann, möglichst gering zu halten. Insbesondere Systeme mit physikalischer Speicheradressierung neigen zur Fragmentierung, wie im Abschnitt 2.2.20 auf Seite 63 ausführlicher beschrieben wird. Gerade in der Entwicklung von embedded Systemen, die häufig nicht auf eine Betriebssystem-gestützten Speicherverwaltung aufbauen, kann die Eindämmung der Fragmentierung über Erfolg und Misserfolg eines Projekts entscheiden.

Eine Antwort auf das Problem der Fragmentierung kann das Vorallokieren größerer Speicherbereiche sein, die dann viele kleine Objekte aufnehmen.

Trennt man in einem solchen Pooling Objekte unterschiedlicher Größe voneinander und hält sie in separaten Pools, kann ein solches Poolingverfahren für eine bestimmte Software feinjustiert werden, indem man die Poolgrößen variiert. Wenn man sich dabei neben der Größe noch auf die Typenunterscheidung verlässt, kann man für spezifische Objekte und deren Allokationsverhalten spezielle Strategien des Poolings anpassen. Man bekommt also in der OO-Programmierung ein zusätzliches Hilfsmittel zur Hand, um ein erfolgreiches Pooling zu betreiben.

Im Folgenden wird ein einfaches Poolingverfahren vorgestellt, das auf die Justierung des Entwicklers angewiesen ist. Es macht sich die Typeninformation der Objekte zunutze, indem verschiedene Pools für verschiedene Objekttypen und nicht einfach nur für Blockgrößen gebildet werden. Damit das Verfahren an beliebige Klassentypen anpassbar ist, basiert es auf der Templatetechnik von C++. Der Templateparameter entspricht dem jeweiligen Klassentyp, der in einem Pool verwaltet werden soll. Je nach zu erwartender Häufigkeit von Allokationen und Deallokationen, beziehungsweise nach der durchschnittlichen Anzahl allokierter Objekte eines Typs, kann nun die Größe eines solchen Pools angepasst werden. Ziel eines Pools sollte sein, die häufigsten Allokationen für einen Typ abzufangen und nur wenige für die freie Heapallokation übrig zu lassen. Damit führt man auch eine „Best-Fit-Strategie“[13] für den entsprechenden Typ ein. Da der Pool nur Objekte gleicher Größe zusammenfasst, implementiert man damit auch die optimale Best-Fit-Strategie. Da der Pool die freien Blöcke in einer separaten Liste führt, muss auch nicht im eigentlichen Wortsinne nach einem passenden Bereich „gesucht“ werden. Die Poolimplementierung spart dadurch auch Laufzeit.

Listing 6.41. Einfaches Objektpooling

```
#include <cstddef>
#include <cstdlib>

template <typename T>
struct ObjFrameBlock
{
  typedef unsigned char byte;
  typedef T element_type;
  byte data[ sizeof(T) ];
};
```

[13] Die Best-Fit-Strategie bezeichnet die Vorgehensweise, den am besten passenden freien Speicherbereich für einen neu zu allokierenden Block zu finden. Demgegenüber steht die „First-Fit-Strategie“, die sich mit dem ersten freien Bereich begnügt, in den der neu zu allokierende Block passt. Die Best-Fit-Strategie versucht den Speicher optimal zu nutzen, während die First-Fit-Strategie die Laufzeit – den Rechenaufwand für die Speicherallokation – reduziert.

```
template <typename T>
struct ObjFrame : public ObjFrameBlock<T>
{
  ObjFrame() : next(NULL), prev(NULL) {}
  void* getData() { return ObjFrameBlock<T>::data; }
  ObjFrame *next, *prev;
};
template <typename T, unsigned int s = 100>
class ObjPool
{
public:
  typedef T           element_type;
  typedef ObjFrame<T> frame_type;
  ObjPool();
  ~ObjPool();

  bool isIn( const void *p ) const
  {
    return    buffer[0].getData() <= p
           && p <= buffer[poolsize-1].getData();
  }
  void *getMemForObj()
  {
    void *ret = NULL;
    if( free )
    {
      frame_type *f = free;
      frame_type *n = free->next;
      if( n ) n->prev = NULL;
      free = n;

      f->next = used;
      if( used ) used->prev = f;
      used = f;

      ret = f;
      n++;
    }
    return ret;
  }
  void freeMemOfObj( void * );

  std::size_t numberObjects() const { return n; }
```

```
private:
  frame_type *buffer;
  frame_type *free;
  frame_type *used;
  std::size_t n;
  enum { poolsize = s };
};

template <typename T, unsigned int s>
ObjPool<T,s>::ObjPool() : n(0), free(NULL), used(NULL)
{
  buffer = new frame_type[poolsize];
  for( int i = 0; i < poolsize; i++ )
  {
    frame_type *f = &buffer[i];
    if( free )
    {
      free->prev = f;
    }
    f->next = free;
    free = f;
  }
}

template <typename T, unsigned int s>
ObjPool<T,s>::~ObjPool()
{
  delete [] buffer;
}

template <typename T, unsigned int s>
void ObjPool<T,s>::freeMemOfObj( void *p )
{
  typedef ObjFrameBlock<element_type> block_type;
  if( isIn( p ) )
  {
    n--;
    block_type *b = reinterpret_cast<block_type*>(p);
    frame_type *f = static_cast<frame_type*>(b);

    frame_type *p = f->prev;
    frame_type *n = f->next;
    if( f == used ) used = n;
```

```
    if(p) p->next = n;
    if(n) n->prev = p;

    f->next = free;
    if( free ) free->prev = f;
    free = f;
  }
}
```

Die Anbindung eines Typs an seinen Pool erfolgt über das Überschreiben der Operatoren `new` und `delete`. Die im Listing 6.42 gezeigten Überladungen greifen auf die Standardimplementierungen von `new` und `delete` zurück, wenn der Pool für die Klasse `X` gefüllt ist. Zu beachten ist auch, dass die Operatoren `new` und `delete` in C++ vererbt werden. Man muss also auch den Fall vorsehen, dass diese Operatoren im Kontext eines abgeleiteten Typs aufgerufen werden und damit eine ganz andere Objektgröße allokieren wollen – wie es in Kapitel 6.1.5 mit Listing 6.34 beschrieben ist. In diesem Fall wird die Allokation an die Standardoperatoren weitergereicht.

Listing 6.42. Beispielklasse `X` mit überladenen `new`- und `delete`-Operatoren

```
// Datei: X.h
class X
{
public:
  X() {}
  ~X() {}

  static void *operator new( std::size_t );
  static void operator delete( void * );
};

// Datei: X.cpp
// Hier kann die Poolgröße variiert werden.
ObjPool<X,300> xpool;

void *X::operator new (std::size_t s)
{
  void *p;
  // Es kann sich um einen abgeleiteten Typ handeln.
  if( s != sizeof(X) )
    p = ::operator new(s);
  else
  {
    p = xpool.getMemForObj();
```

```
    if( !p ) // Wenn der Pool schon voll ist.
      p = ::operator new( s );
  }
  return reinterpret_cast<X*>(p);
}

void X::operator delete( void *p )
{
  if( xpool.isIn(p) )
    xpool.freeMemOfObj(p);
  else
    ::operator delete( p );
}
```

Ein solches Pooling kann nun für die Objekttypen eingeführt werden, die im Sinne der Fragmentierung Probleme machen. Das sind häufig kleine Objektgrößen mit sehr vielen nicht-symmetrischen Allokationen und Deallokationen. Das Pooling für alle Typen in einem System einzuführen ergibt deshalb keinen Sinn, da es den Vorteil der dynamischen Allokation, den Heap zur Laufzeit mehrfach zu verwenden, aufgeben würde. Die dynamische Allokation spart ja gerade Speicher, da dieser immer nur so lange belegt wird, wie es für einen bestimmten Vorgang notwendig ist. Einer Optimierung des Fragmentierungsverhaltens sollte also immer eine Messung vorangehen, um festzustellen, für welche Objekttypen ein Optimierungsbedarf besteht.

6.3.2 Nebenläufige Programmierung

Moderne Software hat häufig mehr als nur einen Ablaufstrang. Das heißt, um die Rechenkapazitäten des Computersystems effektiv auszunutzen, schreibt man Programme so, dass unabhängige Programmabläufe parallel zueinander existieren. So kann zum Beispiel ein solcher Ablaufstrang im Vordergrund auf Benutzereingaben warten, während ein Ablaufstrang im Hintergrund für den Benutzer nicht wahrnehmbar eine Datenbank reorganisiert. Der Strang im Vordergrund benötigt keine Prozessorlast, solange der Benutzer nichts eingibt, und wenn der Benutzer aktiv wird, wird auch nur sehr wenig Rechenkapazität gebraucht, um die Eingaben entgegenzunehmen und die Daten weiterzuleiten. Die Fähigkeit eines Systems, parallele Verarbeitung zu ermöglichen, nennt man Multithreading, Multiprozessing oder Multithreading. Die Ablaufstränge selbst, von denen hier die Rede ist, haben unterschiedliche technische Namen. Zwei davon sind allgemein gebräuchlich und sollen hier erklärt werden: *Prozesse* und *Threads*. Der Begriff *Task* hat weniger technischen Hintergrund. Er bezeichnet einfach eine Aufgabe, die ein Programm ausführt. Wie

das System diese Aufgabe ausführt, wird in dem Begriff Task nicht angesprochen. Hier kommen die beiden Begriffe Prozess und Thread ins Spiel.

Prozess

Mit einem Prozess bezeichnet man einen unabhängigen Ablaufstrang, der einen eigenen Speicherpool besitzt, sodass er vor einem direkten Zugriff durch fremden Code geschützt ist. Die Voraussetzung für Multiprozessing auf einem System ist also, dass das System eine virtuelle Speicherverwaltung besitzt. In einer virtuellen Speicherverwaltung werden Speicherräume voneinander abgegrenzt, indem unterschiedliche Adressräume gebildet werden, die unterschiedlichen Prozessen zugeordnet werden können. Wenn ein Prozess auf einen Adressraum zugreifen möchte, der ihm nicht zugeordnet ist, wird er durch das kontrollierende Betriebssystem an diesem Zugriff gehindert. So können Prozesse ungestört durch andere Prozesse ihre Arbeit verrichten. Ob Prozesse für eine eigene Software genutzt werden können, hängt davon ab, ob das Betriebssystem Prozesse unterstützt. Es gibt also weder in C noch in C++ Sprachmerkmale oder Standardbibliotheken, die es ermöglichen mit Prozessen zu arbeiten.

Thread

Ein Thread ist etwas leichtgewichtiger als ein Prozess. Er ist ein ebenso unabhängiger Ablaufstrang, hat aber keinen eigenen Speicherraum. Er teilt sich den Speicher mit anderen Threads im System. Einerseits können sich Threads also gegenseitig beeinflussen. Auf der anderen Seite ist es leichter Daten auszutauschen, da man nicht auf Betriebssystemmittel wie Dateien, Named Pipes[14] oder Shared Memory[15] zurückgreifen muss. Es ist daher relativ leicht, mehrere Threads zu starten und Daten austauschen zu lassen. Schwerer dabei ist, diesen Austausch der Daten ohne Synchronisationsfehler zu organisieren. Auch für die Threads bieten C und C++ keine Sprachmerkmale und Standardbibliotheken[16]. Der Einsatz von Threads hängt, wie der von Prozessen auch, von den Fähigkeiten des Betriebssystems ab.

Insbesondere in der systemnahen Programmierung hat man es häufig mit Nebenläufigkeit in C- oder C++-Code zu tun. Auch wenn diese Sprachen Nebenläufigkeit nicht direkt unterstützen, hat man einiges zu beachten, wenn man sie in einem solchen Umfeld einsetzt. Ein erstes Beispiel in Listing 6.43 soll einige Problembereiche beleuchten:

[14] Named Pipes sind benannte Datenströme, die zum Datenaustausch zwischen Prozessen oder auch unterschiedlichen Systemen eingesetzt werden können.

[15] Shared Memory ist gemeinsam genutzter Speicher, der von mehreren Prozessen aus beschrieben und gelesen werden kann. Die Virtuelle Speicherverwaltung definiert dazu einen Adressraum, der den entsprechenden Prozessen zugänglich gemacht wird.

[16] In Java ist das anders. Java kennt direkte Synchronisationsmechanismen für Threads und hat dafür sogar ein Schlüsselwort. Die Programmierung von Threads wird durch die Standardbibliothek von Java ermöglicht.

Listing 6.43. Erstes (fehlerhaftes) Beispiel zur Threadprogrammierung

```
#define STRICT
#include <windows.h>
#include <stdio.h>

extern volatile long n = 0;

DWORD WINAPI ThreadFunc( LPVOID param )
{
  long i;
  for( i = 0; i < 1000000000; ++i )
  {
    n++;
  }
  return 0;
}

int main( void )
{
  HANDLE hTh1;
  HANDLE hTh2;
  hTh1 = CreateThread( 0, 0, ThreadFunc, 0, 0, 0 );
  hTh2 = CreateThread( 0, 0, ThreadFunc, 0, 0, 0 );

  Sleep( 5000 );

  WaitForSingleObject( hTh1, INFINITE );
  WaitForSingleObject( hTh2, INFINITE );
  CloseHandle( hTh1 );
  CloseHandle( hTh2 );

  printf( "Der Wert von n: %d\n", n );

  return 0;
}
```

In Listing 6.43 wird ein Beispiel für Windows gezeigt. Auch wenn vergleichbare Beispiele auf anderen Systemen strukturell sehr ähnlich aussehen, sind die Funktionsnamen durch die Programmierschnittstelle des Betriebssystems festgelegt. In Listing 6.43 sind es die Funktionen `CreateThread()`, `Sleep()`, `WaitForSingleObject()` und `CloseHandle()`. Diese und die dazugehörigen Typen (`HANDLE`) und Konstanten (`INFINITE`) stehen in der Includedatei `windows.h`. Dieses Beispiel würde also auf einem anderen System sich

an genau diesen Stellen unterscheiden müssen, wo die Windows-spezifischen Funktionsaufrufe verwendet werden.

Aber nun zum Ablauf des Beispielprogramms: es werden zwei Threads erzeugt. Das Erzeugen erledigt die Funktion `CreateThread()`, indem sie beim Betriebssystem einen neuen Thread anmeldet und einen Funktionszeiger entgegennimmt, der auf eine Startfunktion zeigt. Wenn der Thread durch das Betriebssystem aufgesetzt ist, läuft er in die Startfunktion – im Beispiel die Funktion `ThreadFunc()`. Läuft der Thread aus dieser Funktion wieder heraus, wird er vom Betriebssystem beendet. In Listing 6.43 wird die Funktion `ThreadFunc()` von beiden Threads verwendet. Das ist grundsätzlich kein Problem, denn jeder Thread bekommt einen eigenen Aufrufstack. Alle lokalen Variablen in einer Aufruffolge werden also für jeden Thread neu angelegt. Beide Threads greifen im Beispiel auf eine mit Null initialisierte Variable zu und erhöhen sie schrittweise um eine Milliarde. Die Erhöhungsschritte der Threads zusammengenommen müssten also die Zahl 2.000.000.000 ergeben. Welche Zahl wirklich ausgegeben wird, ist allerdings vom Compiler abhängig: Das Compilat, das mit dem MS Visual C++ 7.1-Compiler erzeugt wurde, gibt auf einem Pentium IV mit 1,6 GHz beispielsweise die folgende Ausgabe aus: `Der Wert von n: 1686191631`. Dabei kann der erreichte Wert sehr unterschiedlich sein. Er variiert bei jedem Programmlauf zwischen ≈1.200.000.000 und ≈1.990.000.000. Ähnliche Ergebnisse erreicht man mit dem GCC 3.4.4-Compiler unter Windows. Der Grund dafür liegt in der Tatsache, dass die Inkrementoperation `n++` nicht *atomar* ist. Das heißt, dass es mehrerer Arbeitsschritte durch den Prozessor bedarf um diese Operation zu Ende zu bringen. Der Assemblercode, der durch den Visual C++-Compiler erzeugt wurde, sieht für die Inkrementoperation von `n` so aus:

```
mov ecx, DWORD PTR _n
add ecx, 1
mov DWORD PTR _n, ecx
```

Diese drei Zeilen repräsentieren die drei Arbeitsschritte, die durch den Compiler gemacht werden müssen, um `n` um eins zu erhöhen: Es muss der aktuelle Wert von `n` aus dem Arbeitsspeicher gelesen werden und in einem Prozessorregister gespeichert werden. Danach kann der Wert um eins erhöht werden. Anschließend wird der neue Wert zurück in den Arbeitsspeicher geschrieben.

Es gibt also zwei Stellen, an denen die Gesamtoperation unterbrochen werden kann. Ein paralleler Thread kann entweder zwischen der Lade- und Inkrementoperation oder zwischen der Inkrement- und der Entladeoperation Einfluss nehmen. Im Fall der Beispiels aus Listing 6.43 beeinflussen sich die beiden Threads kräftig gegenseitig, denn sie arbeiten beide auf der gleichen Variablen. Dieses Beispiel soll also als einfaches Negativbeispiel dienen, an dem man sich die Problematik des gemeinsamen Datenzugriffs vor Augen führen kann.

Im nächsten Beispiel in Listing 6.44 soll der Zugriffskonflikt aufgelöst werden. Es wird dazu ein Synchronisationsmechanismus eingesetzt, der wie die Funktionen zum Programmieren von Threads auch Betriebssystem-spezifisch ist.

Listing 6.44. Zweites Beispiel zur Threadprogrammierung

```
#define STRICT
#include <windows.h>
#include <stdio.h>

volatile long n = 0;
CRITICAL_SECTION cs;

DWORD WINAPI ThreadFunc( LPVOID param )
{
  long i;
  for( i = 0; i < 1000000000; ++i )
  {
    EnterCriticalSection( &cs );
    n++;
    LeaveCriticalSection( &cs );
  }

  return 0;
}

int main( void )
{
  HANDLE hTh1, hTh2;
  InitializeCriticalSection( &cs );
  hTh1 = CreateThread( 0, 0, ThreadFunc, 0, 0, 0 );
  hTh2 = CreateThread( 0, 0, ThreadFunc, 0, 0, 0 );

  Sleep( 5000 );

  WaitForSingleObject( hTh1, INFINITE );
  WaitForSingleObject( hTh2, INFINITE );
  CloseHandle( hTh1 );
  CloseHandle( hTh2 );
  DeleteCriticalSection( &cs );

  printf( "Der Wert von n: %d\n", n );

  return 0;
}
```

Unter der Windows-API[17] ist die Critical Section so ein Synchronisationsmechanismus zwischen Threads. Man muss eine Critical Section initialisieren und auch wieder abbauen. Die Funktionen `EnterCriticalSection()` und `LeaveCriticalSection()` wirken wie eine Art Ampel, die einen Thread exklusiv in den Codeabschnitt eintreten lässt, der vor parallelem Zugriff geschützt werden muss. Wenn ein weiterer Thread in den Codeabschnitt eintreten möchte, solange noch ein anderer sich darin befindet, stoppt die Funktion `EnterCriticalSection()` die Ausführung dieses Threads. Erst wenn der kritische Codeabschnitt wieder frei ist, läuft der wartende Thread in diesen hinein. Das führt dazu, dass jeder Thread seine Aufgabe in einem kritischen Codeabschnitt vollständig durchführen kann, bevor ein anderer Thread Zugriff auf diesen Codeabschnitt erhält.

Alle Betriebssysteme, die Multithreading unterstützen, haben vergleichbare Synchronisationsmechanismen, die man analog zu dem Beispiel in Listing 6.44 anwenden kann. In der Multithreadingprogrammierung muss man auf jeden Fall ein gutes Gespür entwickeln, wo solche gemeinsamen Datenzugriffe stattfinden können. Ein häufiges Problem ist beispielsweise die Standardbibliothek. Ob man zum Beispiel auf die Standardausgabe zugreifen kann, ohne sie durch entsprechende Maßnahmen zu sichern, hängt davon ab, ob die Bibliothek für den Multithreadingeinsatz geschrieben wurde. Der Standard fordert es nicht. Mit den meisten Entwicklungssystemen ist es per Voreinstellung auch nicht der Fall. Man kann Bibliotheken für den Multithreadingeinsatz compilieren. Jedoch sollte man sich genau damit beschäftigen, welche Details damit gesichert – synchronisiert – sind und welche nicht.

Die Anwendung des Schlüsselworts `volatile`

Traditionell bringt man den Modifikator `volatile` an Variablen an, die von unabhängigen Ablaufsträngen, wie zum Beispiel Threads oder Interruptroutinen, verwendet werden. Damit unterdrückt man die aggressive Optimierung der Zugriffe auf die entsprechenden Variablen. Eine solche Optimierung könnte dazu führen, dass, statt der Werte in der Speicherstelle der Variablen, zwischengespeicherte Werte verwendet werden. Damit würden die unabhängigen Stränge des Programms mit unterschiedlichen Werten für eine Variable arbeiten. Das Schlüsselwort `volatile` schützt also nicht vor parallelem Zugriff durch mehrere Threads, sondern *erzwingt* sogar einen Zugriff. Dieser Zwang garantiert, dass die unterschiedlichen Ablaufstränge mit den identischen Werten arbeiten, wenn sie auf die gleiche Variable zugreifen.

Wie kann man dieses Prinzip auf Objektzustände ausdehnen? Zunächst einmal ist das Problem bei Objekten nicht ganz so virulent, wie es bei einfachen Variablen der Fall sein kann. Es sind aber durchaus Situationen vorstellbar, in denen Objektzustände in optimierter Weise bearbeitet werden und

[17] API = Application Programmer Interface.

eventuell nicht immer in den Speicher geschrieben beziehungsweise daraus gelesen werden. So kann zum Beispiel das Inlining von Methoden dafür sorgen, dass beim Aufruf nicht ein anderer Stackkontext geschaffen wird (es findet kein Funktionsaufruf statt). Wenn der Stackkontext der gleiche bleibt, kann der Compiler wieder optimieren. Man kann deshalb auch Objekte mit der Speicherklasse `volatile` anlegen. Allerdings müssen dann alle Methoden das Versprechen einhalten, dass immer vollständig auf den Speicher zugegriffen wird. Diese müssen dann mit der Spezifikation `volatile` versehen werden. Natürlich können Attribute in einem Objekt auch die Speicherklasse `volatile` besitzen. Das ist aber etwas anderen als die Spezifikation an einer Methode, die den `this`-Zeiger modifiziert und damit die Zugriffsgarantie für das ganze Objekt abgibt.

Da es sich hier um eine eher selten angewandte Technik handelt, wird der geneigte Leser sicher verstehen, dass diese an einem ebenso seltenen Problembeispiel demonstriert werden soll. Bestimmte Eventdispatcher arbeiten auf der Basis von Ringpuffern. Das sind Arrays, die Eventobjekte aufnehmen können, um sie für den Weitertransport und für die Abarbeitung zu speichern. Ist ein Event abgearbeitet, wird es aus dem Array entfernt. Ringpuffer nennt man ein solchen Array dann, wenn die neuen Elemente immer hinten nach den alten angestellt werden, bis das Ende des Arrays erreicht ist. Danach beginnt man wieder am Anfang des Arrays. Dabei ergibt sich eine Organisationsform, die man bildlich als kreisförmiges Array darstellen könnte. Ich verwende den Ringpufferindex deshalb als Beispiel, weil er technisch sehr einfach ist.

Um einen Ringpuffer zu indizieren, kann man eine ganzzahlige Variable verwenden. Man muss dabei aber immer auf den Überlauf Acht geben. Wenn also die Speicherstelle mit dem höchsten Indexwert erreicht ist, muss man wieder beim ersten anfangen. Damit diese Überprüfung nicht an mehreren verschiedenen Stellen im Code gemacht werden muss, kann man eine Klasse implementieren, die einen Ringpufferindex darstellt. Dazu kapselt man einfach eine ganzzahlige Variable in eine Klasse und definiert den Inkrement- und den Dekrementoperator. Zur Rückkonvertierung in einen ganzzahligen Wert kann man einen Cast-Operator nach `unsigned int` definieren. Damit kann man Objekte der Ringpufferklasse direkt in den Indexoperatorklammern `[]` verwenden.

Listing 6.45. Die Ringindex-Klasse

```
class RingIndex
{
public:
  explicit RingIndex( unsigned int s )
  : max(s), actual(0) {}

  void inc()
```

```
  {
    if( actual == (max-1) )
      actual = 0;
    else
      actual++;
  }

  RingIndex& operator++()
  {
    inc();
    return *this;
  }

  RingIndex operator++(int)
  {
    RingIndex tmp(*this);
    inc();
    return tmp;
  }

  operator unsigned int () const { return actual; }

private:
  unsigned int actual;
  const unsigned int max;
};
```

Auf die Implementierung des Dekrementoperators habe ich zur Vereinfachung verzichtet. Er wird analog zum Inkrementoperator definiert. Natürlich können darüber hinaus noch weitere Methoden wie zum Beispiel Zuweisungs- und Vergleichsoperatoren definiert werden, um der Ringindex-Klasse eine sinnvolle Funktionalität zu geben.

Mit der Implementierung in Listing 6.45 ist die Klasse für einen ersten praktischen Test fertig und kann überprüft werden:

Listing 6.46. Testcode für die Ringindex-Klasse

```
#include <iostream>

int main()
{ using namespace std;

  int array[3];
  RingIndex r(3);
```

```
  array[r++] = 701;
  array[r++] = 702;
  array[r++] = 703;

  cout << array[r++] << endl;
  cout << array[r++] << endl;
  cout << array[r++] << endl;
  cout << array[r++] << endl;
  cout << array[r++] << endl;
  cout << array[r]   << endl;

  return 0;
}
```

Die Ausgabe des Programms

```
701
702
703
701
702
703
```

zeigt, dass die Klasse verwendet werden kann. Da alle Methoden inline definiert sind, können die Methoden sehr gut optimiert werden. Letztendlich wird nicht mehr Code erzeugt als beim direkten Einsatz einer Variablen, die man selbst kontrollieren muss.

Zurück zum Anwendungsfall des Ringpuffers: es ist denkbar, dass eine Indizierung eines Ringpuffers von unterschiedlichen Ablaufsträngen aus geschieht. Vielleicht schreibt ein Thread munter in den Ringpuffer hinein und ein anderer liest. Das wirft genau zwei Probleme auf:

1. Die Zugriffe müssen synchronisiert werden.
2. Die Werte dürfen nicht in einem Cache oder in Registern zwischengespeichert werden. Auch nicht durch einen optimierten Code, den der Compiler erzeugen kann.

Obwohl das erste Problem das wichtigere ist, lasse ich es in diesem Abschnitt außen vor! Es wird im Abschnitt 6.3.2 ab Seite 382 besprochen.

In diesem Abschnitt möchte ich das zweite Problem besprechen, das sich mit der vollständigen Abarbeitung abgeschlossener Operationen befasst[18].

In der Ringindex-Klasse sind die Methoden inline definiert, was zu einer aggressiven Optimierung durch den Compiler führen kann. Diese Optimierung kann unterbunden werden, wenn das Ringindex-Objekt mit der Speicherklasse `volatile` instanziiert wird.

[18] Insbesondere dann, wenn diese nicht atomar sind.

Listing 6.47. `RingIndex`-Objekt mit der Speicherklasse `volatile`

```
...
int array[3];
volatile RingIndex r(3);

array[r++] = 701; // Fehler, Operator ++ undefiniert!
...
```

Das Beipiel wird durch den Compiler nicht mehr übersetzt, denn es werden Methoden aufgerufen, die nicht garantieren, dass alle verwendeten Objektattribute vollständig gelesen und geschrieben werden. Diese Garantie muss nun gegeben werden. Dazu gibt es in C++ eine Konstruktion, die der `const`-Spezifikation an Methoden gleicht. Man schreibt das Schlüsselwort `volatile` hinten an die Methodendeklaration und garantiert damit die geeignete Behandlung des `this`-Zeigers:

Listing 6.48. Allgemeine Syntax zur `volatile`-Spezifikation

```
class X
{
public:
  void f() volatile;
};

void X::f() volatile
{
  // ...
}

int main()
{
  volatile X obj;

  obj.f();

  return 0;
}
```

Dabei ist die `volatile`-Spezifikation kompatibel mit der von `const`. Die `const`-Spezifikation besagt schließlich nur, dass ausschließlich gelesen werden soll. Die `volatile`-Spezifikation trägt dazu bei, dass `wirklich` gelesen werden soll. Eine Methode kann also beide Spezifikationen tragen.

Listing 6.49. Allgemeine Syntax zur `const`- und `volatile`-Spezifikation

```
class X
{
public:
  void f() const volatile;
};

void X::f() const volatile
{
  // ...
}

int main()
{
  volatile X obj;

  obj.f();

  return 0;
}
```

Beide Spezifikationen (`const` und `volatile`) werden dazu herangezogen, die Signatur einer Methode zu bestimmen. Methoden mit und ohne Spezifikation haben also unterschiedliche Signaturen.

Wenn die Ringindex-Klasse also als `volatile`-Objekt anwendbar sein soll, muss man die Methoden mit eben dieser Spezifikation versehen. Das zieht noch ein paar Konsequenzen nach sich: der Postinkrementoperator ++ muss ein Objekt der Klasse zurückliefern, das mit einem Copy-Konstruktor instanziiert wird. Da der Copy-Konstruktor normalerweise seinen Parameter nicht mit `volatile` maskiert, muss ein solcher Copy-Konstruktor definiert werden:

Listing 6.50. Die Ringindex-Klasse mit `volatile`-Spezifikationen

```
class RingIndex
{
public:
  explicit RingIndex( unsigned int s )
  : max(s), actual(0) {}

  RingIndex( const volatile RingIndex& r )
  : max(r.max), actual(r.actual) {}

  void inc() volatile
  {
    if( actual == (max-1) )
      actual = 0;
```

```
    else
      actual++;
  }

  volatile RingIndex& operator++() volatile
  {
    inc();
    return *this;
  }

  RingIndex operator++(int) volatile
  {
    RingIndex tmp(*this);
    inc();
    return tmp;
  }

  operator unsigned int () const volatile
  { return actual; }

private:
  unsigned int actual;
  const unsigned int max;
};
```

Mit dieser Definition der Klasse lässt sich ein `volatile`-Objekt der Ringindex-Klasse instanziieren und auch verwenden. Wie an dem Copy-Konstruktor schon gezeigt wurde, können beliebige Parameter mit `volatile` maskiert werden. Wenn ein mit `volatile` instanziiertes Objekt einer Funktion oder einer Methode übergeben werden soll, muss der entsprechende Parameter maskiert sein. Genau so, wie es analog für die `const`-Spezifikation gilt.

Manche STL-Klassen definieren Methoden[19], die ausschließlich mit einer vorhandenen und einer nicht vorhandenen `const`-Spezifikation versehen sind. Damit werden für den Aufruf an konstanten Objekten unterschiedliche Funktionalitäten oder unterschiedliche Rückgabetypen realisiert.

Analog dazu kann man Methoden auch mit der `volatile`-Spezifikation voneinander unterschieden. Wenn die Ringindexklasse in den einfachen Fällen optimiert werden soll und in anderen nicht, könnte sie zum Beispiel zwei vollständige Sätze aller Methoden enthalten. Der eine Satz trägt die Spezifikation `volatile`, der andere nicht.

Das in diesem Abschnitt besprochene Thema ist sehr speziell und eine praktische Anwendung ist entsprechend selten. Das hat unter anderem auch den Grund, dass Code in systemnahen Bereichen bis heute häufig in C geschrieben wird.

[19] Wie zum Beispiel die Methoden `begin()` und `end()` an den Containerklassen.

Literatur

[Alexandrescu01] Andrei Alexandrescu: Modern C++ Design. Generic Programming and Design Patterns Applied. Reading MA 2001.

[CzarneckiEisenecker00] Krzystof Czarnecki, Ulrich W. Eisenecker. Generative Programming. Methods, Tools and Applications. Reading MA 2000.

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. Reading MA 1995.

[ISO-C++] International Standard ISO/IEC 14882. Programming Languages – C++. First edition 1998-09-01. New York 1998.

[C++-Templates] Nicolai Josuttis: C++ Templates.

[Coplien98] James O. Coplien: Multi-Paradigm Design for C++. Reading 1998.

[Koenig97] Andrew Koenig, Barbara Moo: Ruminations on C++. A Decade of Programming Insight and Experience. Reading 1997.

[Lippman02] Stanley B. Lippman, Josée Lajoie: C++ Primer. Deutsche Übersetzung. 1. Auflage Bonn 2002.

[Meyers98] Scott Meyers: Effektiv C++ programmieren. 3., aktualisierte Auflage. Bonn[u. a.] 1998.

[Meyers99] Scott Meyers: Mehr effektiv C++ programmieren. Bonn, Paris [u.a.] 1999.

[Meyers01] Scott Meyers: Effective STL. 50 Specific Ways to Improve Your Use of the Standard Template Library. 2001.

[Stroustrup98] Bjarne Stroustrup: Die C++ Programmiersprache. 3. aktualisierte Auflage. Bonn 1998.

Abbildungsverzeichnis

2.1 Allokation und Deallokation bei physikalischer Speicherverwaltung 64
2.2 Fragmentierter Speicher 65
2.3 Einfache Vererbung 131
2.4 Mehrfachvererbung 133
2.5 Objektlayout bei Mehrfachvererbung 137
3.1 Der Iterator 178
3.2 Das Zustandsmuster 180
3.3 Das Zustandsdiagramm des Ampelbeispiels 182
3.4 Die Organisation einer doppelt verketteten Liste 199
5.1 Logische Elementsequenz bei der Verwendung von Reverse-Iteratoren 295
5.2 Das halb offene Intervall bei der Verwendung von `begin()` und `end()` 296
5.3 Organisationsprinzip des Binären Baums 300
5.4 Schematische Darstellung der Beziehungen zwischen den Iteratortypen 310
5.5 Einfache Organisation eines Valarrays 332
5.6 Der lineare Aufbau eines Valarray-Objekts 339
5.7 Die Matrizendarstellung 339
5.8 Die Elementauswahl durch ein `slice`-Objekt 339
5.9 Eine weitere Elementauswahl durch ein `slice`-Objekt 340
5.10 Die 2 × 2-Submatrix aus der 3 × 4-Matrix 341

Listings

2.1 Erstes Beispiel 9
2.2 Verzweigungen im Präprozessorlauf 15
2.3 Verzweigungen im Präprozessorlauf anhand einer Definition 16
2.4 Die Anwendung des Präprozessoroperators `defined` 16
2.5 Beispiel: bedingte Compilierung 17
2.6 Beispiel: bedingte Compilierung 17
2.7 `#ifndef` anders ausgedrückt 17
2.8 Detektion zur Compilezeit 18
2.9 Abbruch des Compilerlaufs bei Fehlerfall 18
2.10 Die Anwendung des Präprozessoroperators `##` 20
2.11 Anwendung des `sizeof`-Operators 24
2.12 Syntax des `enum`-Typs 27
2.13 Zeigersyntax 31
2.14 Zeigersyntax 32
2.15 Zeigersyntax 32
2.16 Zeigersyntax 32
2.17 Zeigertypenkonflikt 32
2.18 C-Strings 33
2.19 Array-Syntax 34
2.20 Array-Syntax 34
2.21 Array-Syntax 35
2.22 Zeigerinkrementierung auf einem Array 35
2.23 Indizierung eines Arrays über einen Zeiger 36
2.24 Zugriff auf ein Array mittels Zeigerarithmetik 36
2.25 Referenzsyntax 37
2.26 Implizite Typenkonvertierung 37
2.27 Explizite Typenkonvertierung 38
2.28 Dynamische Speicherallokation mit Hilfe von Zeigern 39
2.29 Implizite Typenkonvertierung in einem Ausdruck 40
2.30 Der Bitoperator `&` 42
2.31 Deklaration und Anweisungen 44
2.32 Einfache Funktionen 50
2.33 Einfache Funktion 51
2.34 Funktionsdefinition 52
2.35 Funktionsprototyp 52
2.36 Funktionsprototyp mit benanntem Parameter 52

2.37 Inklusionsbeziehungen zur Sicherung des passenden Funktionsaufrufs 52
2.38 Ein Funktionszeiger 53
2.39 Aufruf der Funktion über den Funktionszeiger durch die Klammern 54
2.40 Überladene Funktion `minimum()` 55
2.41 Normale Funktion 56
2.42 Inlinefunktion 56
2.43 Definition und Anwendung eines Makros 58
2.44 Dynamische Allokation von Speicher in C 60
2.45 Sicherung der dynamischen Speicherallokation 61
2.46 Zugriff auf den allozierten Speicher 61
2.47 Dynamische Speicherallokation in C++ mit `new` und `delete` . 62
2.48 Dynamische Allokation eines einzelnen Objekts 62
2.49 Eine einfache Struktur 67
2.50 Eine verschachtelte Struktur 68
2.51 Zugriffe auf Strukturelemente 68
2.52 Anwendung des Pfeiloperators 68
2.53 Übergabe eines Strukturzeigers an eine Funktion 69
2.54 Const-Maskierung eines Strukturzeigers 69
2.55 Noch einmal der Punkt 70
2.56 Eine einfache Klasse 71
2.57 Zugriffsversuch auf die Elemente der Klasse 71
2.58 Öffnen der Sichtbarkeit der Attribute 71
2.59 Klassenattribute 72
2.60 Die drei Möglichkeiten der Konstantendefinition in Klassen . 73
2.61 Methoden 73
2.62 Implementierung von Methoden 74
2.63 Methodenzugriff 74
2.64 Der `this`-Zeiger 75
2.65 Inlining von Methoden 76
2.66 Die Sichtbarkeiten von Klassenelementen 77
2.67 Standardmäßige private Vererbung 78
2.68 Öffentliche Vererbung 78
2.69 Ein Punkt in einem zweidimensionalen Koordinatensystem . . 79
2.70 Die Definition eines Konstruktors 79
2.71 Eine Initialisierungsliste 80
2.72 Fehlende Initialisierungsliste 81
2.73 Korrekte Initialisierungsliste 82
2.74 Initialisierung von Basisklassenelementen 82
2.75 Initialisierung eines konstanten Attributs 83
2.76 Initialisierung eines konstanten Attributs 83
2.77 Initialisierung eines Referenzobjekts 84

2.78 Deklaration eines Destruktors 84
2.79 Definition eines Destruktors 85
2.80 Speicherallokation im Konstruktor und Freigabe im Destruktor 85
2.81 Der Standardkonstruktor 86
2.82 Der Copy-Konstruktor 87
2.83 Verwendung des Copy-Konstruktors bei der Initialisierung . . 88
2.84 Copy-Konstruktoraufruf bei Werteübergabe an eine Funktion 88
2.85 Typischer Fehler im Zusammenhang mit dem Copy-Konstruktor 89
2.86 Privater Copy-Konstruktor 90
2.87 Implizite Typenkonvertierung mit dem Konstruktor 92
2.88 `explicit`-Deklaration eines Konstruktors 93
2.89 Unterbundene implizite Konvertierung 93
2.90 Automatische Instanziierungen 95
2.91 Dynamische Instanziierung 97
2.92 Dynamische Instanziierung nach AT&T-Standard 98
2.93 Die `nothrow`-Variante des `new`-Operators 99
2.94 Statische Variable in einer Funktion 103
2.95 Ein `mutable`-Attribut 103
2.96 Schleife ohne Einfluss auf die Schleifenbedingung 105
2.97 Mögliches Ergebnis einer Compileroptimierung 106
2.98 Zuordnung einer Methode zu einer Klasse 106
2.99 Globale Funktion 107
2.100 Anwendung des Scope-Operators beim Ansprechen von Bezeichnern 107
2.101 Qualifizierung durch den Scope-Operator 107
2.102 Qualifizierung bei Mehrfachvererbung 108
2.103 ADL versus Qualifizierung 109
2.104 Verschachtelte Klasse 110
2.105 Instanziierung einer verschachtelten Klasse 110
2.106 Private verschachtelte Klasse 110
2.107 Eine `friend`-Klasse 111
2.108 Eine `friend`-Funktion 112
2.109 Eine `friend`-Methode 112
2.110 Instanzzählung mit statischen Elementen 113
2.111 Vererbung 114
2.112 Verweis eines Kindklassenobjekts mit einem Basisklassenzeiger 115
2.113 Sichtbarkeit in der Vererbung 116
2.114 Elementsichtbarkeit bei standardmäßig privater Vererbung . . 117
2.115 Elementsichtbarkeit bei öffentlicher Vererbung 118
2.116 Elementsichtbarkeit bei geschützter Vererbung 119

2.117 Elementsichtbarkeit bei privater Vererbung 120
2.118 Einfache Methode 122
2.119 Methodenaufruf 123
2.120 Virtuelle Methode 123
2.121 Rein virtuelle Methode 123
2.122 Implementierung einer rein virtuellen Methode in einer Kindklasse 124
2.123 Virtueller Destruktor 125
2.124 Typische Elemente einer Basisklasse 126
2.125 Rein virtueller Destruktor 126
2.126 Gültige Konvertierungen mit `static_cast<>()` 128
2.127 Fehlende `const`-Spezifikation 129
2.128 Korrekturcast bei fehlender `const`-Spezifikation 130
2.129 Dynamischer Downcast 131
2.130 Dynamischer Sidecast 133
2.131 Mehrfachvererbung 136
2.132 Mehrfachvererbung mit Attributen 136
2.133 Adressverschiebung bei Upcast 137
2.134 Rautenförmige Vererbungshierarchie 138
2.135 Virtuelle Vererbung 139
2.136 Dynamische Verschiebung der Objektadresse beim dynamischen Upcast 140
2.137 Initialisierung einer virtuellen Basisklasse 140
2.138 Const-Maskierung 143
2.139 Auswirkung der Const-Maskierung 144
2.140 Konstanter Rückgabewert 145
2.141 Komplexe Zahl als Strukturtyp 145
2.142 Überladener Operator 145
2.143 Anwendung eines überladenen Operators 146
2.144 Ausgabestreamoperator 146
2.145 Anwendung überladener Operatoren 147
2.146 Verkettung der überladenen Streamoperatoren 148
2.147 Überladener Operator als Klassenelement 149
2.148 Kopie des Rückgabewertes 150
2.149 Kopie des Rückgabewertes 150
2.150 Überladung des Subtraktionsoperators 150
2.151 Überladung des Multiplikationsoperators 150
2.152 Überladung des Zuweisungsoperators 151
2.153 Der überladene Dereferenzierungsoperator 152
2.154 Globaler Dereferenzierungsoperator 152
2.155 Überladener Pfeiloperator 153
2.156 Allgemeine Form eines Typenkonvertierungsoperators 153
2.157 Typenkonvertierungsoperator nach `bool` 154

2.158 Überladener Indexoperator 154
2.159 Überladener Klammeroperator 155
2.160 Exception bei fehlgeschlagener Allokation 155
2.161 Mehrere `catch`-Blöcke 156
2.162 Hierarchische Exceptionhandler 157
2.163 Ausnahmebehandlung anhand der Typunterscheidung 159
2.164 Weiterreichen einer teilbehandelten Ausnahme 160
2.165 Implementierung der Methode `what()` 160
2.166 Allokation von Speicher 162
2.167 Öffnen von Dateien 162
2.168 Eine einfache Dateiklasse 163
2.169 Abbruch des Konstruktors mit einer Exception 164
2.170 Schematische Darstellung eines Exception-sicheren Destruktors 164
2.171 Implementierung einer Logfile-Klasse mit Exceptions 166
2.172 Schema einer Fehlerbehandlung in einem Konstruktor 168
2.173 Sicherung dynamisch allokierter Objekte durch Autopointer . 169
3.1 Implementierung des Ampelbeispiels 183
3.2 Singletonimplementierung nach [GoF95] 188
3.3 Singleton mit privatem Kopierkonstruktor 189
3.4 Singleton mit privatem Standardkonstruktor 190
3.5 Singleton mit statischer Instanz 191
3.6 Singleton mit Wächterklasse 192
3.7 Template-Implementierung der Elternklasse(n) 194
3.8 Verwendung eines Singletons auf Makrobasis 195
3.9 Definition eines auf Templates basierenden Wächters 195
3.10 Eine 1-zu-3-Komposition 197
3.11 Eine 1-zu-5-Aggregation 198
3.12 Liste Knoten und Element 200
3.13 Die Implementierung des Knotens 200
3.14 Die Attribute der Liste 200
3.15 Gesicherte Copy-Konstruktoren 201
3.16 Methoden der Liste 201
3.17 Die Zugriffsklasse 202
3.18 Die Zugriffsschnittstelle der Liste 202
3.19 Implementierung der Zugriffsklasse 203
3.20 Anwendung der Liste 204
3.21 Eine Basisklasse für polymorphe Listenelemente 204
3.22 Die Implementierung der Methode `AddEnd()` für polymorphe Listenelemente 205
3.23 Verschachtelte Klassen in der Listenimplementierung 205
3.24 Die Implementierung der Liste 206
3.25 Eine Vektor-Implementierung 208

3.26 Einsatz von überladenen Operatoren für den Elementzugriff . 211
3.27 Typenkonvertierungsoperator für Gültigkeitsprüfung 212
3.28 Operatoren für Dereferenzierung, Inkrement und Dekrement in der Zugriffsklasse 212
3.29 Implementierung einer Vektormethode mit Invariantenprüfung 215
3.30 Invariantenprüfung durch entfernbaren Code 215
3.31 Invariantenprüfung mit dem Makro `assert()` 216
3.32 Eine separate Methode für den Invariantencheck 216
3.33 Ein Template für Assertions 219
3.34 Anwendung der Assertion-Templates 219
3.35 Ein verfeinertes Assertion-Template 219
4.1 Eine traditionelle Implementierung der Funktion `min()` 223
4.2 Eine weitere traditionelle Implementierung von `min()` 223
4.3 Überladene Versionen von `min()` 223
4.4 `min()` als Makroimplementierung 224
4.5 Unerwünschter Nebeneffekt bei der Verwendung eines Makros 224
4.6 Template-Implementierung der Funktion `min()` 224
4.7 Die alte Template-Syntax mit dem Schlüsselwort `class` 225
4.8 Referenzübergabe und Rückgabe bei Templatefunktion 225
4.9 Template-Implementierung einer Inline-Funktion 225
4.10 Ein Klassentemplate 226
4.11 Instanziierung eines Klassentemplates 226
4.12 Dynamische Instanziierung eines Objekts einer Templateklasse 227
4.13 Methoden in einem Klassentemplate 227
4.14 Separate Templates für die Methodenimplementierungen . . . 228
4.15 Anwendung der Template-Methoden 228
4.16 Syntax der expliziten Templateinstanziierung 229
4.17 Instanziierung eines Templates durch eine Typendefinition . . 229
4.18 Deklaration und Implementierung eines Membertemplates . . 230
4.19 Vollständige Spezialisierung einer Templateklasse 231
4.20 Allgemeines Template 232
4.21 Partielle Spezialisierung eines Templates 233
4.22 Anwendung von Grundtemplate und spezialisiertem Template 233
4.23 Methodentemplate mit partieller Spezialisierung 234
4.24 Vollständige Spezialisierung des Methodentemplates 234
4.25 Vorgabeargument in der Templateparameterliste 235
4.26 Qualifizierung über den `this`-Zeiger 236
4.27 Nichtqualifiziertes Einsetzen eines Operators 236
4.28 Qualifizierter Aufruf eines Operators 237
4.29 Überschriebene Methode 237

4.30 Aufruf einer überschriebenen Basisklassenmethode durch Qualifizierung . . . 237
4.31 Uneindeutiges Compileergebnis bei fehlender Qualifizierung in Templates . . . 238
4.32 Eindeutige Qualifizierung in Templates . . . 240
4.33 Das `rebind`-Element aus der STL . . . 241
4.34 Qualifizierung des `rebind`-Templates . . . 242
4.35 Der Problemcode . . . 242
4.36 Implementierung einer `friend`-Funktion in einer Klasse . . . 243
4.37 Syntaktischer Fehler . . . 243
4.38 Die Lösung . . . 244
4.39 Alte Syntax . . . 244
4.40 Neue Syntax . . . 244
4.41 Nicht standardkonformer Template-Code . . . 245
4.42 Standardkonformes Anzeigen eines Typs . . . 246
4.43 Template-Templateparameter . . . 247
4.44 Template-Templateparameter . . . 247
4.45 Verschachtelte Template-Templateparameter . . . 247
4.46 Vorgabetypen in Template-Templateparametern . . . 248
4.47 Reduzierte Form . . . 248
4.48 Beispiel eines generischen Mappings . . . 249
4.49 Anpassung von Template-Templateparametern für ein bestimmtes Übergabetemplate . . . 250
4.50 Der Test des Mappings . . . 251
4.51 Eine Templateimplementierung der verketteten Liste . . . 251
4.52 Die Anwendung der Liste . . . 253
4.53 Eine Templateimplementierung des Vektors . . . 254
4.54 Ein Template-basierter Smart Pointer . . . 256
4.55 Ein Referenzzähler . . . 257
4.56 Implementierung des Smart Pointers . . . 257
4.57 Implementierung von Vergleichsoperatoren für den Smart Pointer . . . 259
4.58 Der Test der Smart-Pointer-Klasse . . . 260
4.59 Smart Pointer in einem Container . . . 261
4.60 Teilimplementierung eines Template-basierten Vektors . . . 262
4.61 Allgemeines Klassentemplate . . . 264
4.62 Explizite Instanziierung einer Templateklasse . . . 264
4.63 Explizite Instanziierung einer Template-Methode . . . 264
4.64 Ungültige doppelte Instanziierung . . . 265
4.65 Reine Deklarations-Templates . . . 265
4.66 Implementierungs-Templates . . . 266
4.67 Instanziierung und Verwendung der Typeninformation . . . 266
4.68 Instanziierung der Implementierungen . . . 267

4.69 Verwendung des Schlüsselworts `export` 268
4.70 Compilezeitbedingungen mit Templates 269
4.71 Typendetektion zur Compilezeit 269
4.72 Rechenanweisungen für den Compiler 269
4.73 Compilezeit-Assertion aus [Alexandrescu01] 273
4.74 Die Template-Parameter des Strings in der Standard-Library . 274
5.1 Einfache Anwendung von C++-Streams 280
5.2 Überladen des Shift-Operators zur Verwendung als Ausgabeoperator 281
5.3 Anwendung des definierten Ausgabeoperators auf verschiedene Streams 281
5.4 Definition und Anwendung eines Eingabeoperators 282
5.5 Whitespace löschende Wirkungsweise des Eingabeoperators . 283
5.6 Anwendung der Methode `get()` zum Einlesen aller Zeichen . 284
5.7 Formatierung einer Ausgabe auf ein `ostream`-Objekt über Flags 285
5.8 Definition einer Feldbreite in einer Ausgabe 285
5.9 Formatierung über Manipulatoren 286
5.10 Ein Manipulator mit einem Parameter 287
5.11 Feldbreite und Leerraumformatierung mit Manipulatoren . . . 287
5.12 Definition eines eigenen Manipulators 288
5.13 Einfaches Kopierprogramm mit File-Streams 289
5.14 Anwendung eines `ostringstream`-Objekts als Textpuffer 290
5.15 Anwendung des `istringstream` zur Eingabe aus Strings 291
5.16 Anwendung von Iteratoren in der STL 294
5.17 Kurze Demonstration der STL-Liste 296
5.18 Eine Demonstration des STL-Vektors 297
5.19 Variierung des Datencontainers im fachlichen Code 297
5.20 Drei mögliche Zugriffsarten auf den Vektor 298
5.21 Ein Demonstrationsbeispiel zum STL-Set 299
5.22 Ein Demobeispiel zur STL-Map 301
5.23 Ein Programm zum Zählen von Wörtern in Texten 301
5.24 Ein Demobeispiel zum STL-Stack 303
5.25 Ein Demobeispiel zur STL-Queue 305
5.26 Ein Demobeispiel zur STL-Priority Queue 307
5.27 Erstes Beispiel zu einem Stream-Iterator 312
5.28 Ein weiterer Wörterzähler mit Streamiteratoren 313
5.29 Umformatierung von Streams über Iteratoren 314
5.30 Der Algorithmus `for_each` 315
5.31 Das Prädikat als Klassenobjekt 316
5.32 Variable Parametrisierung eines Prädikats 317
5.33 Der Algorithmus `count_if` 318
5.34 Funktorimplementierung des Prädikats 318

5.35 Adaptieren von Prädikaten 319
5.36 Standardkonforme Deklaration eines Prädikats 320
5.37 Standardkonforme Deklaration eines Prädikat-Templates . . . 320
5.38 Ein in der STL vordefiniertes Prädikat 321
5.39 Eine Modifikation des Prädikats 321
5.40 Eine Umformung 322
5.41 Verändernder Algorithmus `transform()` 322
5.42 Anwendung von `transform` auf einen C-String 323
5.43 Transformation und Anhängen an neuen Container 323
5.44 Rechenoperationen auf Mengen mit dem Algorithmus `transform` 324
5.45 Der Algorithmus `copy` 325
5.46 Inputiterator durch `back_inserter` 325
5.47 Eigene Implementierung eines Algorithmus 326
5.48 Iteratorzugriff auf ein `std::string`-Objekt 326
5.49 Verschiedene Konstruktoren der Stringklasse 327
5.50 Anwendung von `string::npos` 327
5.51 Initialisierung aus einer Containersequenz 328
5.52 Definition einer auf `wchar_t` basierenden Stringklasse 329
5.53 Definition einer Stringklasse mit anderen Eigenschaften . . . 330
5.54 Einfache Verwendung eines Valarrays 333
5.55 Vergrößerung des Valarrays durch Aufruf von `resize()` 333
5.56 Arithmetische Operationen auf Valarrays 335
5.57 Mengenoperationen auf Valarrays 336
5.58 Submengenbildung durch Indizierung mit `slice`-Objekten . . 337
5.59 Veränderung der Werte einer `slice`-Auswahl 340
5.60 Eine Submatrix mit `gslice` 340
5.61 Automatische Freigabe des dynamisch instanziierten Objekts . 342
5.62 Lösen des Objekts aus der Verantwortung des Autopointers . . 343
5.63 Vorgezogenes Löschen des Objekts am Autopointer 343
5.64 Automatisches Reset durch Neuzuweisung des Autopointers . 343
5.65 Die Operatoren `*` und `->` am Autopointer 344
6.1 Sicherung gegen Mehrfachinklusion 348
6.2 Die Alles-Überall-Inklusion 350
6.3 Einhaltung einer Inklusionsreihenfolge 350
6.4 Die Deklaration der Klasse `DBZugriff` 352
6.5 Vorwärtsdeklaration und Verwendung der Klasse `DBZugriff` . 352
6.6 Einbindung der Typeninformation erst bei Zugriff auf Klassenelementen von `DBZugriff` 353
6.7 Syntax der Namensräume 354
6.8 Erweiterbarkeit von Namensräumen 354
6.9 Namensräume über mehrere Headerdateien 355
6.10 Verschachtelte Namensräume 355

6.11 Anwendung der `using`-Direktive auf ein Einzelelement 356
6.12 Anwendung der `using`-Direktive auf einen Namensraum . . . 356
6.13 Integration eines Elementes aus einem fremden Namensraum 357
6.14 Integration eines ganzen Namensraums 357
6.15 Die Aliasdefinition 357
6.16 Erweiterung des Namens bei Elementdefinition 357
6.17 Programm im AT&T-Stil 358
6.18 Programm im ISO-C++-Stil 358
6.19 Globale Öffnung des Namensraums `std` 359
6.20 Lokale Öffnung des Namensraums `std` 359
6.21 Schritt 1: Ein sehr einfaches Programm 361
6.22 Schritt 2: Woher kommt der Ausgabeoperator? 362
6.23 Schritt 3: Der Operator « kommt aus `std` 362
6.24 Logische Folge aus den Schritten 1 - 4 363
6.25 Das Template `min()` 364
6.26 Die Benutzung eines Templates in mehreren Modulen 365
6.27 Inklusion einer Klassendeklaration mit V-Tabelle in mehreren Modulen 365
6.28 Überladener `new`-Operator 368
6.29 Überladener `new`-Operator mit Fehlerroutine 368
6.30 Überladener `delete`-Operator 369
6.31 Explizite statische Deklaration von `new`- und `delete`-Operatoren 369
6.32 Deklarationspaare von `new`- und `delete`-Operatoren 370
6.33 Demo zur Überladung von `new` und `delete` 371
6.34 Operator `new` mit Sicherungsabfrage 372
6.35 Operator `delete` mit Sicherheitsabfrage 373
6.36 Schema des globalen `new`-Operators aus [Meyers98] 373
6.37 Schematische Deklaration zur Persistenz 374
6.38 Schematische Implementierung zur Persistenz 375
6.39 Einlesende Funktion 375
6.40 Konstruktor zum Einlesen eines Objekts 376
6.41 Einfaches Objektpooling 378
6.42 Beispielklasse `X` mit überladenen `new`- und `delete`-Operatoren 381
6.43 Erstes (fehlerhaftes) Beispiel zur Threadprogrammierung . . . 384
6.44 Zweites Beispiel zur Threadprogrammierung 386
6.45 Die Ringindex-Klasse 388
6.46 Testcode für die Ringindex-Klasse 389
6.47 `RingIndex`-Objekt mit der Speicherklasse `volatile` 391
6.48 Allgemeine Syntax zur `volatile`-Spezifikation 391
6.49 Allgemeine Syntax zur `const`- und `volatile`-Spezifikation . . 392
6.50 Die Ringindex-Klasse mit `volatile`-Spezifikationen 392

Index

++, 294
#, 20
##, 20
#define, 14, 58, 348
#elif, 15
#else, 15
#endif, 15
#error, 18
#ifdef, 16
#ifndef, 16, 348
#if, 15
#include, 14
#pragma, 19
#undef, 18
__DATE__, 20
__FILE__, 20
__LINE__, 20
__TIME__, 20
__cplusplus, 16, 17, 20
<iomanip>, 287
<sstream>, 290
NDEBUG, 218
assert(), 217, 218
at(), 298
auto_ptr<>, 171, 342, 344
auto, 101
back_insert_iterator<>, 311
back_inserter<>, 323
bad_alloc, 170
basic_string<>, 274, 275
basic_string, 328
begin(), 293, 295–297, 299
bind1st(), 319
bind2nd(), 319
bool, 23
cerr, 280
char, 23
cin, 280
class, 71
clog, 280
const_cast<>(), 127
const_iterator, 293, 311
const_reverse_iterator, 294
const, 105, 129, 141, 293, 311
const-Maskierung, 143
copy<>(), 324
copy_backward<>(), 324
copy_if<>(), 324
count_if<>, 321
count_if, 317
cout, 280
delete, 62, 97
deque, 292
double, 23
do-while(), 47
dynamic_cast<>(), 127
else, 46
end(), 293, 295–297, 299
enum, 27
equal_to<>, 321
exception, 169
explicit, 93
extern, 102
first, 301
float, 23
for(), 47
for_each<>(), 316, 326
for_each, 317
free(), 60
friend, 111, 242, 243
friend-Deklaration, 111
front_insert_iterator<>, 311
greater<>, 321
greater_equal<>, 321
gslice, 340
if(), 46

ifstream, 289
inline, 56
insert_iterator<>, 311
int, 23
istream_iterator<>, 312
istringstream, 290
iterator, 293
key_type, 301
less<>, 321
less_equal<>, 321
list, 292, 295
long, 23
malloc(), 60
map, 292, 301
multimap, 292
multiset, 292
mutable, 103
namespace std, 353, 358
namespace, 354
negate<>(), 323
new, 62, 97
new Variante mit nothrow, 99
not_equal_to<>, 321
nothrow, 99
ofstream, 289
operator «, 146
ostream_iterator<>, 312
ostringstream, 290
out_of_range, 170
pair<>, 301
priority_queue, 293, 302, 306
qset, 292
queue, 293, 302, 305
rbegin(), 294, 311
register, 101
reinterpret_cast<>(), 127
rend(), 294, 311
reset(), 343
reverse_iterator, 294, 311
second, 301
set, 299
short, 23
sizeof, 18, 24
slice_array, 340
slice, 337
stack, 293, 302, 303
static_cast<>(), 93, 127
static, 102, 112
std::exception, 160, 166
stringstream, 290
string, 170, 274, 326
struct, 67
switch(), 48
template<template T», 247
template, 222
this-Zeiger, 75
transform(), 323
transform<>(), 324
typedef, 297
typename, 244, 246
unsigned, 23
using namespace, 356
using, 356
valarray<>, 332
value_type, 301
vector, 292, 297
virtual, 121, 138
volatile, 105, 387
wchar_t, 275
what(), 170
while(), 46

Ableitung, 114
ADL, 109, 361
Algorithmen, 315
Allokator, 248
Anweisungen, 44
Argumentenabhängiges Lookup-Verfahren, 361
Arithmetische Operatoren, 41
Arrays, 29
Aspektorientierte Programmierung, 273
Assertions, 218, 272
Attribute, 72
Aufzählungstypen mit enum, 27

Ausdrücke, 39
Ausnahmebehandlung, 155
Ausnahmeklassen, 169
auto_ptr<T>, 168
Autopointer, 168, 171, 342

Barton-Nackman-Trick, 242
Bedingungen, 45
Bezeichner, 12
Bidirectional-Iteratoren, 309
Bitoperatoren, 41

C-Strings, 33
Compilezeitverhalten, 264
Conditions, 45
Const-Iterator, 311
Container Adapter, 302
Container mit Templates, 251
Containerklassen, 197
Containertypen, 292
Copy-Konstruktor, 87
Copy-Konstruktor-Problem, 89

Datencontainer, 197, 292
Default-Konstruktor, 85
defined, 16
Dekomposition, 271
Delegationsprinzip, 72, 75, 76
Dereferenzierungsoperator, 152
Design-Patterns, 176
Destruktor, 84
Dynamische Speicherallokation, 59
Dynamischer Polymorphismus, 124

Einfüge-Iteratoren, 311
Entwurfsmuster, 176
Exception Handling, 155
Exception-Klassen, 169
Exceptions, 155, 169
Exceptions in Konstruktoren, 165
Explizite Template-
    instanziierung, 264
Explizite Typen-
    konvertierung, 38, 93

Forward-Iteratoren, 309
Fragmentierung, 63
Funktionen, 49
Funktionsdeklaration, 51
Funktionsinlining, 56
Funktionsprototyp, 51
Funktionssignatur, 54
Funktionstemplates, 223
Funktionsüberladung, 54
Funktionszeiger, 53
Funktoren, 315

Generische Programmierung, 221

Headerdateien, 348

Idiome, 176, 271
Implizite Typen-
    konvertierung, 37, 93
Include-Dateien, 348
Initialisierungsliste, 80
Inklusionsmodell, 263, 265
Inline-Methode, 76
Input-Iteratoren, 309
Insert-Iteratoren, 311
Instanziierung von Objekten, 94
Invariante, 213, 214
Iterator, 178
Iterator-Muster, 178
Iterator-Tags, 310
Iteratorarten, 309
Iteratoren, 293, 308

Kapselung, 76
Klassenbegriff, 174
Klassenkonzept, 69
Klassenmethoden, 73
Klassentemplates, 226
Koenig-Lookup-Verfahren, 361
Konstante Attribute, 72
Konstante Rückgabewerte, 145
Konstanten, 26
Konstrollstrukturen, 45
Konstruktor, 79

Konvertierungsoperator, 93
Kopierkonstruktor, 87
Kopierkonstruktor-Problem, 89

L-Value, 40, 43
L-Wert, 40, 43
Linker, 363
Liste, 198
Literalkonstanten, 25
Logische Operatoren, 44

Makros, 58
Manipulatoren, 286
Maskierung mit `const`, 143
Mehrfachvererbung, 134
Member-Templates, 230
Methoden, 73
Methodentemplates, 227
Minimalgröße von Objekten, 372
Multiprozessing, 382
Multitasking, 382
Multithreading, 382
Muster, 176, 271

Nachbedingungen, 217
Namemangling, 354
Namensräume, 353
Nebenläufige
    Programmierung, 382

Objektinstanziierung, 94
Objektlayout, 136
Objektorientierte
    Programmierung, 173
Objektpooling, 377
Operator `!=`, 45
Operator `()`, 154
Operator `*`, 152
Operator `->`, 152
Operator `::`, 106
Operator `<=`, 45
Operator `<`, 45
Operator `==`, 45
Operator `=`, 151
Operator `>=`, 45
Operator `>`, 45
Operator `[]`, 152
Operator `delete`, 97
Operator `new`, 97
Operatoren, 40
Operatorüberladung, 145, 149
Output-Iteratoren, 309

Partielle Spezialisierung, 232
Patterns, 176
Persistenz, 374
Pointer, 29
Pointerarithmetik, 34
Policy Based Design, 270
Polymorphismus, 121
Präprozessor, 13
Präprozessormakros, 58
Prototyp, 51
Prozess, 382, 383
Prüfcode, 217
Pseudocontainer, 332

Qualifizierung, 107, 236
Qualifizierung in Templates, 241

R-Value, 40, 43, 54
R-Wert, 40, 43, 54
Random-Access-Iteratoren, 309
Referenzen, 36
Reverse-Iterator, 311

Scope-Operator, 106
Separationsmodell, 267
Serialisierung, 374
Sichtbarkeit, 77, 116
Sichtbarkeit der Vererbung, 78, 116
Signatur, 54, 144
Singleton, 187
Singleton-Muster, 187
Smartpointer, 255
Speicherfragmentierung, 63, 210
Speicherklassen, 100
Spezialisierung von Templates, 231

Stack, 22
Standard-Library, 277
Standard Template Library, 292
Standardbibliothek, 277
Standarddatentypen, 22
Standardkonstruktor, 85
Standardnamensraum, 353
Statemachine, 179
Statische Attribute, 112
Statische Methoden, 112
Statischer Polymorphismus, 124
STL, 292
STL Algorithmen, 315
Stream-Iteratoren, 312
Streams, 280
Stringklasse, 274, 326
Struktur, 67
Suffixe, 25
Symbolische Konstanten, 20
Systemprogrammierung, 377

Template-Spezialisierung, 231
Template-Template-Parameter, 247
Template-Vorgabeargumente, 235
Templateinstanziierung, 229
Templates, 222
Textpuffer, 290
Thread, 382, 383
Typecast, 37
Typenkonvertierung, 37, 91
Typenkonvertierung mit Konstruktoren, 91
Typenkonvertierungsoperator, 127

Überladen von Klammeroperatoren, 154
Überladung arithmetischer Operatoren, 149
Überladung des Zuweisungsoperators, 151
Überladung von `delete[]`, 370
Überladung von `delete`, 367, 368, 370
Überladung von `new[]`, 370
Überladung von `new`, 367, 368, 370
Überladung von Typenkonvertierungsoperatoren, 153

Variablen, 21
Vektor, 208
Vererbung, 114
Vergleichsoperatoren, 43, 45
Verkettete Liste, 198
Verschachtelte Klassen, 110
Virtuelle Methoden, 121
Virtuelle Speicherverwaltung, 383
Virtuelle Vererbung, 138
Virtueller Destruktor, 125
Vorbedingungen, 217
Vorgabeargumente für Templates, 235

Zeiger, 29
Zeigerarithmetik, 34, 36
Zugriffsoperatoren, 152
Zustand, 179
Zustandsmaschine, 179
Zustandsmuster, 179
Zuweisungsoperatoren, 43