Jürgen Wolf

CD-ROM mit

- ✓ Openbooks
- ✓ Quellcode der Beispiele
- ✓ Entwicklungsumgebungen

C++ von A bis Z

Das umfassende Handbuch

- Das Lehr- und Nachschlagewerk
- Für Einsteiger: ANSI C++ verstehen und anwenden
- Für Profis: UML, Netzwerkprogrammierung, GUI- und Multimedia-Bibliotheken

2., aktualisierte Auflage

Galileo Computing

Jürgen Wolf

C++ von A bis Z

Das umfassende Handbuch

Galileo Press

Liebe Leserin, lieber Leser,

Jürgen Wolf veröffentlichte bereits mehrere Bücher bei Galileo Computing. Vielleicht kennen Sie ja schon eines davon, z. B. »C von A bis Z«, »Shell-Programmierung« oder »Linux-UNIX-Programmierung«? All diese Werke sind von hoher fachlicher Qualität, vermitteln das jeweilige Themengebiet sehr gründlich und umfassend und sind zudem sehr unterhaltsam geschrieben. Diese Merkmale treffen auch auf sein Buch zur C++-Programmierung zu.

Als die erste Auflage dieses Buches erschien, war es angesichts der Vielzahl an C++-Literatur aus Verlagssicht zumindest ein gewagtes Unterfangen, ein weiteres Buch zu diesem Thema herauszubringen. Doch die Qualität dieses Werkes hat sowohl uns als auch die Leser sofort überzeugt: Es ist schlicht besser als viele andere Bücher zum selben Thema.

Sowohl ein Programmieranfänger (mit guten PC-Kenntnissen) als auch ein »ausgewachsener« C++-Entwickler wird dieses Werk mit Gewinn lesen. Die einen werden Schritt für Schritt vorgehen, die anderen suchen gezielt nach Themen oder Lösungen. Und beide werden nicht enttäuscht werden, denn sie finden hier eine praktische Einführung in die Sprache ebenso wie ausführliches Fachwissen zur Standard Template Library, Boost, zur Socket- oder GUI-Programmierung. Doch selbst auf fast 1300 Seiten kann C++ nicht erschöpfend erklärt werden. Vielleicht fehlt Ihnen Grundlagenwissen zu C? Auch das finden Sie im Buch, nämlich auf der beiliegenden Buch-CD. Dort stellen wir das komplette Werk »C von A bis Z« zur Verfügung, als leicht navigierbare HTML-Version.

Und jetzt wünsche ich Ihnen viel Spaß beim Lesen!

Ihre Judith Stevens-Lemoine
Lektorat Galileo Computing

judith.stevens@galileo-press.de
www.galileocomputing.de
Galileo Press · Rheinwerkallee 4 · 53227 Bonn

Auf einen Blick

Der Name Galileo Press geht auf den italienischen Mathematiker und Philosophen Galileo Galilei (1564–1642) zurück. Er gilt als Gründungsfigur der neuzeitlichen Wissenschaft und wurde berühmt als Verfechter des modernen, heliozentrischen Weltbilds. Legendär ist sein Ausspruch *Eppur se muove* (Und sie bewegt sich doch). Das Emblem von Galileo Press ist der Jupiter, umkreist von den vier Galileischen Monden. Galilei entdeckte die nach ihm benannten Monde 1610.

Gerne stehen wir Ihnen mit Rat und Tat zur Seite:
judith.stevens@galileo-press.de bei Fragen und Anmerkungen zum Inhalt des Buches
service@galileo-press.de für versandkostenfreie Bestellungen und Reklamationen
julia.bruch@galileo-press.de für Rezensions- und Schulungsexemplare

Lektorat Judith Stevens-Lemoine, Anne Scheibe
Fachgutachten Martin Conrad
Korrektorat Marlis Appel, Troisdorf
Typografie und Layout Vera Brauner
Herstellung Katrin Müller
Satz Typographie & Computer, Krefeld
Druck und Bindung Bercker Graphischer Betrieb, Kevelaer

Dieses Buch wurde gesetzt aus der Linotype Syntax Serif (9,25/13,25 pt) in FrameMaker. Gedruckt wurde es auf chlorfrei gebleichtem Offsetpapier.

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.

ISBN 978-3-8362-1429-2

2., aktualisierte Auflage 2009

Inhalt

Vorwort

Es ist mir eine große Freude, Ihnen die zweite Auflage dieses Buches präsentieren zu dürfen. Die vorherige Auflage war immerhin schon drei Jahre alt, so dass sie auf den neuesten Stand gebracht werden musste. Einige Kapitel wurden außerdem ein wenig erweitert. Neben der Aktualisierung des Buches wurden auch Fehler der alten Auflage behoben. An dieser Stelle möchte ich mich bei den vielen Lesern der ersten Auflage bedanken, die mir mit Hinweisen, Fehlermeldungen und Vorschlägen geholfen haben, dieses Buch noch weiter zu verbessern.

Über dieses Buch

Mittlerweile gibt es eine ganze Menge von Büchern zur C++-Programmiersprache. Dies allein zeigt schon die Popularität, die diese Sprache genießt. Natürlich habe ich es mir nicht nehmen lassen, auch etwas vom großen Kuchen abzubekommen. Wieso aber habe ich mir die Mühe gemacht, gut acht Monate auf das Schreiben eines weiteren Buches über diese Sprache zu verwenden, wenn schon unzählige davon vorhanden sind? Es gibt schließlich viele hervorragende C++-Bücher für Einsteiger. Und Profis greifen gern zum vielzitierten Stroustrup (»Die C++ Programmiersprache«). Für welche Zielgruppe ist also dieses Buch gedacht, und was bietet es, was andere C++-Bücher nicht haben?

Zielgruppe

Dieses Buch richtet sich an mehrere Zielgruppen und kann auch als Nachschlagewerk verwendet werden. Es eignet sich für den absoluten Einsteiger in die Programmierung genauso wie für den fortgeschrittenen Programmierer. Auch »Umsteiger« von C dürften keine Probleme haben – das Déjà-vu-Erlebnis sollte sich hier in Grenzen halten (speziell auch für die Leser meines Buches »C von A bis Z«). Es ist ebenfalls nicht nötig, bereits über Kenntnisse irgendeiner anderen Programmiersprache zu verfügen. Nach diesem Buch können Sie ohne Bedenken auf das Stroustrup-Buch zurückgreifen und sich selbst den letzten Schliff geben. Natürlich darf man bei einem Buch zur Programmierung (gleich welcher Programmiersprache) immer erwarten, dass der Leser grundlegende Kenntnisse zum Arbeiten mit einem PC mitbringt.

Aber was enthält dieses Buch nun wirklich, was andere Bücher zu C++ nicht bieten? Eine ganze Menge: Neben dem üblichen ANSI-C++-Standard geht dieses Buch auch auf Themen wie STL, Boost, Socket- oder die GUI-Programmierung ein.

C und C++

Dass C++ eine Erweiterung von C ist, bringt so manchen Buchautor zum Grübeln. Soll man jetzt ein Buch zweiteilen und somit C und C++ verwenden, oder soll man C ganz ignorieren? Zugegeben, wer reines C++ programmieren will, benötigt kein C. Aber C völlig außen vor zu lassen und als unnötigen Ballast zu bezeichnen ist ein weiterer folgenschwerer Fehler. Wer das Glück hat und ein neues, »leeres« Projekt anfangen darf, dem kann C egal sein. Aber oft hat man als Programmierer die undankbare Aufgabe, »alte« Programme zu pflegen bzw. zu verbessern. Und häufig sind solche Programme noch in C geschrieben. Hierbei erhält man dann meistens den Auftrag, das Programm objektorientiert und flexibler zu machen, d. h., man soll aus einem in C geschriebenen Programm ein C++-Programm machen. Wie dem auch sei, über das Pro und Contra ließen sich noch viele Zeilen schreiben, und eben aus diesem Grund finden Sie auf der Buch-CD als kostenlose Beigabe die HTML-Version des Buches »C von A bis Z« (2. Auflage).

Betriebssystem

Da dieses Buch über den gewöhnlichen Standard-C++-Umfang hinausgeht, stellen Sie sich sicherlich die Frage, ob Sie das alles auch auf Ihrem Betriebssystem benutzen können. Hierzu ein ganz klares Ja. Alle Themen sind so gut wie plattformunabhängig und wurden auf Linux, UNIX (BSD) und MS Windows getestet. Und wenn es trotzdem mal die ein oder andere »Ungereimtheit« gibt, wird darauf hingewiesen, und es werden entsprechende Alternativen demonstriert. Natürlich ist dieses Buch so aufgebaut, dass zuerst auf die in C++ standardisierten Dinge eingegangen wird.

Übersicht

In **Kapitel 1,** »Grundlagen in C++«, wird auf die reinen Grundlagen von C++ eingegangen. Diese umfassen einfache Dinge wie den Bezeichner, Basisdatentypen, Konstanten, einfache Ein-/Ausgabe, die grundlegenden Operatoren, Kommen-

tare, Kontrollstrukturen wie Verzweigungen oder Schleifen, Funktionen und Präprozessor-Direktiven.

Kapitel 2, »Höhere und fortgeschrittene Datentypen«, geht auf die höheren und fortgeschrittenen Datentypen wie Zeiger, Referenzen, Arrays, Zeichenketten (C-Strings) und Strukturen ein.

Kapitel 3, »Gültigkeitsbereiche, spezielle Deklarationen und Typumwandlungen«, behandelt unspektakuläre, aber sehr wichtige Themen wie die Gültigkeitsbereiche, Namensräume, Speicherklassenattribute, Typqualifikatoren und Typumwandlungen.

Kapitel 4, »Objektorientierte Programmierung«, ist das wichtigste, aber auch schwierigste Kapitel in diesem Buch. Hier werden alle Themen behandelt, die die objektorientierte Programmierung betreffen. Das Verstehen dieses Kapitels ist die Grundlage für die weiteren Kapitel im Buch und für C++ generell. Dabei geht es um die Klassen und wie man diese in der Praxis anwenden kann. Natürlich wird hierbei auch auf Vererbung, Polymorphismus und Mehrfachvererbung eingegangen.

Kapitel 5, »Templates und STL«, enthält dann die Erstellung eigener Funktions- und Klassen-Templates. Darauf basiert ja auch die STL (*Standard Template Library*), weshalb auch sehr umfangreich auf STL eingegangen wird.

Da in den vorangegangenen Kapiteln häufig die Rede von Exceptions (Ausnahmebehandlungen) war, erläutert **Kapitel 6,** »Exception-Handling«, dieses Thema sehr umfassend.

Gewöhnlich besitzt jede Sprache einen Standardumfang. So natürlich auch C++ mit ihren Standardbibliotheken. Zunächst wird sehr ausführlich die String-Bibliothek vorgestellt. Anschließend folgen die Klassen für die Ein-/Ausgabe. Dabei werden neben den gewöhnlichen Klassen für die Ein- bzw. Ausgabe-Streams auch die Klassen für die Datei- und String-Streams behandelt. Auch für Mathematiker hält C++ mit den Klassen `valarray` und `complex` einiges bereit. Natürlich wird auch auf andere numerische Bibliotheken eingegangen. Am Ende von **Kapitel 7,** »C++-Standardbibliothek«, finden Sie zudem eine Beschreibung dazu, wie Sie eine Typerkennung zur Laufzeit durchführen können.

Kapitel 8, »Weiteres zum C++-Guru«, enthält viele Informationen, über die man als C++-Programmierer verfügen sollte. Neben einfacheren Dingen – zum Beispiel, wie Sie eigene Module erstellen – spielen hierbei auch die feinen, aber sehr wichtigen Unterschiede zwischen C und C++ eine Rolle. Des Weiteren finden Sie eine sehr umfassende Einführung in UML und die Erstellung von Klassendiagrammen. Natürlich wird auch ein wenig der Programmierstil vorgestellt. Sehr

selten liest man etwas über Boost, weshalb in Kapitel acht mit der Bibliothek `Boost.Regex` (für reguläre Ausdrücke) darauf eingegangen wird.

Kapitel 9, »Netzwerkprogrammierung und Cross-Plattform-Entwicklung in C++«, behandelt die Netzwerkprogrammierung. Da diese nicht mehr zu den portablen Sachen gehört, wird in diesem Kapitel u. a. auf die Cross-Plattform-Entwicklung eingegangen. Hierzu werden Sie eine eigene Socket-Klasse schreiben.

Im letzten Kapitel – **Kapitel 10,** »GUI- und Multimediaprogrammierung in C++« – erhalten Sie zunächst einen Überblick über die gängigen GUI- und Multimediabibliotheken. Anschließend wird sehr umfassend das wxWidgets-Framework beschrieben. Auch hierbei müssen Sie sich keine Gedanken bezüglich der Portabilität machen. wxWidgets gibt es auf allen gängigen Plattformen, und – noch besser – die Quelltexte lassen sich ohne Änderungen auf den verschiedensten Systemen übersetzen.

[»] In diesem Buch finden Sie an einigen Stellen grau hinterlegte Kästen, in denen Sie weiterführende Informationen zu bestimmten Themen erhalten. Einige Kästen sind am Rand mit einem speziellen Icon gekennzeichnet. Dieses Icon steht neben einem Hinweis-Kasten. Hier erhalten Sie z. B. Informationen zu Unterschieden zwischen C und C++, Hinweise auf Fehlerquellen sowie kleine Tipps und Tricks, die Ihnen das (Programmierer-)Leben erleichtern.

Buch-CD

Auf der Buch-CD finden Sie sämtliche Quellcodes aus dem Buch wieder. Ebenso wurden einige Anleitungen gängiger Compiler (Entwicklungsumgebungen) erstellt, die Ihnen zeigen, wie Sie aus einem Quelltext ein ausführbares Programm machen. Sofern Sie also absoluter Neuling sind, sollten Sie zuerst einen Blick auf diese Buch-CD werfen.

Damit auch MS Windows-Anwender gleich loslegen können, finden diese auf der Buch-CD die Bloodshed-Dev-C++-Entwicklungsumgebung. Als Linux- bzw. UNIX-Anwender hat man es da leichter. Hier befindet sich gleich alles an Board des Betriebssystems und muss nicht extra besorgt werden. Gegebenenfalls kann es sein, dass man einzelne Pakete nachinstallieren muss (abhängig von der verwendeten Distribution).

Neben der bereits erwähnten HTML-Version meines Buches »C von A bis Z« finden Sie auch noch eine weitere HTML-Version des Buches »IT-Handbuch für Fachinformatiker« von Sascha Kersken. Dieses Buch wurde ausdrücklich auf meinen Wunsch hin auf die Buch-CD gepresst. Der Grund dafür ist, dass viele ange-

hende Programmierer häufig die Grundlagen der Informatik einfach überspringen. Solche Wissensdefizite machen sich jedoch irgendwann bemerkbar. Ich verwende außerdem selbst immer wieder gern dieses Buch für meine Recherchen.

Danksagung

Ein Buch mit einem solchen Umfang schreibt man nicht einfach so, und häufig steckt monatelange Arbeit darin (und vom »Autor-Sein« kann man nicht leben). Schlimmer noch muss es aber für die Menschen sein, die mit dem Autor in dieser Zeit zusammenleben. Meine Frau hat wohl schon die Hoffnung aufgegeben, mit einem normalen Menschen zusammen sein zu dürfen. Auch mein Sohn (vier Jahre) erkennt und protestiert sofort, wenn ich mich wieder zum PC »wegschleichen« will. Kurz gesagt: Beim Schreiben von Büchern geht einfach schnell mal die Harmonie flöten. Daher ist hier mal wieder ein riesiges Dankeschön an Euch beide nötig. Ihr seid die wichtigsten Menschen in meinem Leben.

Die wichtigste Person im fachlichen Bereich ist Martin Conrad, der stets für Fachlesungen meiner Bücher zur Verfügung steht. Er ist auch Maintainer »meiner« Webseite, und mittlerweile brüten wir beide auch unser erstes Projekt aus. Obwohl ich Martin mittlerweile mehrere Jahre kenne, haben wir uns noch nie im realen Leben gesehen. Wer sagt da, dass virtuelle Freundschaften nicht funktionieren. Auch dir, lieber Martin, vielen Dank für deine tolle Unterstützung.

Bücher zu schreiben ist das eine, aber einen Verlag, der so flexibel ist und dem Autor so viele Freiheiten lässt, findet man kein zweites Mal. Natürlich gibt es hier auch immer eine Person, die hinter den Kulissen steht und diesen Prozess koordiniert. In diesem Fall ist es meine Lektorin Judith Stevens-Lemoine, die mich jetzt bereits beim vierten gemeinsamen Buchprojekt unterstützt. Vielen Dank, Judith, für die tolle Zusammenarbeit.

Jürgen Wolf

Vorwort des Fachgutachters

C++ – nur eine Erweiterung von C?

Der Geruch eines neuen Buches erfüllt den Raum, Ihr Rechner ist hochgefahren, und Sie sitzen in den Startlöchern, um sich mit C++ zu beschäftigen? Ich hoffe, Sie haben den Kaffee nicht vergessen!

Die meisten Leser werden sich vorher mit C beschäftigt haben, werden also eher Umsteiger auf eine objektorientierte Sprache sein. Hierbei war die Auswahl, was erlernt werden soll, sicher eine schwierige Entscheidung. Wenn erfahrene Programmierer gefragt werden, welche Programmiersprache empfehlenswert ist, gibt es annähernd so viele Antworten wie Personen, die befragt werden.

Falls es Einwände gegen C++ gegeben hat, wird das meist die Nähe zu C gewesen sein. Es ist durchaus möglich, C++ genauso wie C als prozedurale Sprache zu verwenden und dabei die Bibliotheken eher als Zusatzfunktionen zu nutzen, und leider wird C++ auch von vielen Leuten auf diese Art genutzt.

Um wirklich gut zu wartenden objektorientierten Code zu erhalten, ist ein komplettes Umdenken nötig, da die Schwerpunkte und Schwierigkeiten beider Ansätze an völlig unterschiedlichen Stellen liegen.

Bei C war es noch möglich, kleinere Programme einfach zu beginnen und sie wachsen zu lassen – bei objektorientierten Sprachen führt diese Vorgehensweise schnell zu einem Neuschreiben des gesamten Codes, wenn der Code nicht prozedural sein soll, was ja Sinn der Sache ist. Es ist bei der Planung notwendig, den objektorientierten Ansatz zu verstehen und die Möglichkeiten zu kennen, die C++ bietet. Fehlendes Wissen kann hier zu ärgerlichen groben Designfehlern und viel überflüssiger Arbeit führen.

Namensräume, Klassen, Templates, die STL ... – es ist gut, dies alles zu kennen und in die Planung der Programme einzubeziehen.

Aus diesen Gründen möchte ich Ihnen nahelegen, Ihr neues Buch möglichst komplett durchzuarbeiten und zu vermeiden, mit Anfangswissen Projekte zu starten.

Ich wünsche Ihnen viel Spaß mit Ihrem neuen Buch und der Welt der objektorientierten Programmierung!

Martin Conrad

Dieses Kapitel geht auf die Grundlagen der C++-Programmierung bzw. auf die grundlegenden Themen der meisten Programmiersprachen überhaupt ein. Von Basisdatentypen, Standard-I/O-Streams, Konstanten, lexikalischen Elementen, Operatoren, Begrenzern, verschiedenen Kontrollstrukturen und Funktionen bis zum Präprozessor finden Sie hier vieles, was Ihnen in anderen Programmiersprachen recht ähnlich (oder fast gleich) begegnet. Hierbei soll auch kurz auf die Geschichte von C++ und auf die Frage eingegangen werden, wie man eigentlich ein Programm erstellt.

1 Grundlagen in C++

1.1 Die Entstehung von C++

Ursprünglich wurde C++ 1979 von Dr. Bjarne Stroustrup entwickelt, um Simulationsprojekte mit geringem Speicher- und Zeitbedarf zu programmieren. Auch hier lieferte (wie schon bei C) kein geringeres Betriebssystem als UNIX die Ursprungsplattform dieser Sprache. Stroustrup musste (damals noch bei den Bell Labs beschäftigt) den UNIX-Betriebssystemkern auf verteilte Programmierung hin analysieren. Für größere Projekte verwendete Stroustrup bisher die Sprachen Simula – die allerdings in der Praxis recht langsam bei der Ausführung ist – und BCPL, die zwar sehr schnell ist, aber sich für große Projekte nicht eignete.

Simula und BCPL

Simula gilt als Vorgänger von Smalltalk. Viele der mit Simula eingeführten Konzepte finden sich in modernen objektorientierten Programmiersprachen wieder.

BCPL (Kurzform für *Basic Combined Programming Language*) ist eine um 1967 von Martin Richards entwickelte kompilierte, systemnahe Programmiersprache, abgeleitet von der *Combined/Cambridge Programming Language* CPL. Es handelt sich um eine Sprache aus der ALGOL-Familie.

Homepage von Stroustrup

Wer mehr über Bjarne Stroustrup erfahren möchte, findet unter der URL *http://public.research.att.com/~bs/homepage.html* seine Homepage.

Stroustrup erweiterte die Sprache C um ein Klassenkonzept, wofür er die Sprache Simula-67 (mit der Bildung von Klassen, Vererbung und dem Entwurf virtueller Funktionen) und später dann Algol68 (Überladen von Operatoren; Deklarationen im Quelltext frei platzierbar) sowie Ada (Entwicklung von Templates, Ausnahmebehandlung) als Vorlage nahm. Heraus kam ein »C mit Klassen« (woraus etwas später auch C++ wurde). C wurde als Ursprung verwendet, weil diese Sprache schnellen Code erzeugte, einfach auf andere Plattformen zu portieren ist und fester Bestandteil von UNIX ist. Vor allem ist C auch eine weitverbreitete Sprache, wenn nicht sogar die am weitesten verbreitete Sprache überhaupt. Natürlich wurde auch weiterhin auf die Kompatibilität von C geachtet, damit auch in C entwickelte Programme in C++-Programmen liefen. Insgesamt wurde »C mit Klassen« zunächst um folgende Sprachelemente erweitert, auf die später noch eingegangen wird:

- Klassen
- Vererbung (ohne Polymorphismus)
- Konstruktoren, Destruktoren
- Funktionen
- Friend-Deklaration
- Typüberprüfung

Einige Zeit später, 1982, wurde aus »C mit Klassen« die Programmiersprache C++. Der Inkrementoperator ++ am Ende des C sollte darauf hinweisen, dass die Programmiersprache C++ aus der Programmiersprache C entstanden ist und erweitert wurde. Folgende neue Features sind dann gegenüber »C mit Klassen« hinzugekommen:

- virtuelle Funktionen
- Überladen von Funktionen
- Überladen von Operatoren
- Referenzen
- Konstanten
- veränderbare Freispeicherverwaltung
- verbesserte Typüberprüfung
- Kommentare mit // am Zeilenende anfügen (von BCPL)

1985 erschien dann die Version 2.0 von C++, die wiederum folgende Neuerungen enthielt:

- Mehrfachvererbung
- abstrakte Klassen

- statische und konstante Elementfunktionen
- Erweiterung des Schutzmodells um das Schlüsselwort `protected`

Relativ spät, 1991, fand das erste Treffen der ISO(*International Organisation for Standardization*)-Workgroup statt, um C++ zu standardisieren, was 1995 zu einem *Draft Standard* führte.

Draft Standard

Draft Standard ist die Vorstufe zum Standard. Das Protokoll hat die Analyse- und Testphase bestanden, kann jedoch noch modifiziert werden.

Allerdings dauerte es wiederum drei weitere Jahre, bis 1998 C++ dann endlich von der ISO genormt wurde (ISO/IEC 14882:1998). Erweitert wurde C++ in dieser Zeit um folgende Features:

- Templates
- Ausnahmebehandlung
- Namensräume
- neuartige Typumwandlung
- boolesche Typen

Natürlich entstanden während der Zeit der Weiterentwicklung von C++ auch eine Menge Standardbibliotheken wie bspw. die Stream-I/O-Bibliothek, die die bis dahin traditionellen C-Funktionen `printf()` und `scanf()` abgelöst haben. Ebenfalls eine gewaltige Standardbibliothek wurde von HP mit STL (Standard Template Library) hinzugefügt.

Im Jahre 2003 wurde dann die erste überarbeitete Version von ISO/IEC 14882:1998 verabschiedet und ISO/IEC 14882:2003 eingeführt. Allerdings stellt diese Revision lediglich eine Verbesserung von ISO/IEC 14882:1998 dar und keine »neue« Version. Eine neue Version von C++ (auch bekannt unter C++0x bzw. C++200x) soll angeblich noch in diesem Jahrzehnt erscheinen – zumindest deutet der Name dies an.

Seit Ende 2006 wurde als Termin für die Fertigstellung das Jahr 2009 erwähnt. Somit wurde aus C++0x die Abkürzung C++09. Ob der Termin tatsächlich eingehalten wird, steht natürlich wieder auf einem anderen Blatt. Und neben dem Einhalten des Termins müssen auch die Compiler der Hersteller entsprechend erneuert werden. Man kann nur hoffen, dass die Compiler-Hersteller den neuen C++09-Standard konsequent umsetzen werden (beim `g++`, denke ich, wird dies recht schnell gehen), so dass dann bis 2010 bzw. spätestens 2011 alle Compiler

auf den neuen Standard aufbauen. Aber wie gesagt, das ist lediglich Wunschdenken des Autors.

Zu einem der Top-Features im neuen C++09-Standard gehört ganz klar die Unterstützung von Threads. Ein Thema, das früher vom Standardisierungskomitee nie beachtet wurde, muss jetzt im Zeitalter von Mehrprozessorumgebungen einfach standardisiert werden. Der C++09-Standard wird auf jeden Fall eine eigene Bibliothek zur Unterstützung von Threads enthalten.

Auch die Programmbibliothek wird u. a. um reguläre Ausdrücke, Zufallsbibliothek, intelligente Zeiger, neue untergeordnete assoziative Container, Tupel, Werkzeuge für C++-Metaprogrammierung usw. erweitert. Viele dieser Erweiterungen sind Teil der Boost-Bibliothek und werden mit minimaler Veränderung übernommen. Interessant ist auch, dass der C99-Standard in einer für C++ geänderten Form enthalten sein wird.

Auch der Sprachkern soll im C++09-Standard verbessert bzw. vereinfacht werden. Neu hinzukommen werden Konzepte (engl.: *concepts*) zur Spracherweiterung.

[»] **Hinweis**

Weitere Informationen (Weblinks) zum künftigen C++09-Standard habe ich Ihnen auf der Buch-CD zusammengestellt.

1.1.1 Aufbau von C++

C++ ist im Gegensatz zu Sprachen wie Smalltalk oder Eiffel keine reine objektorientierte Sprache – genauer gesagt: Smalltalk bzw. Eiffel wurden ohne Wenn und Aber als objektorientierte Sprachen entwickelt. C++ hingegen entstand ja aus C – womit es sich hierbei um eine Programmiersprache mit objektorientierter Unterstützung handelt, was zum einen Nachteile, zum anderen aber auch Vorteile mit sich bringt (siehe »Stärken von C++« und »Schwächen von C++« weiter unten in diesem Abschnitt).

[»] **Hinweis**

Sofern Sie bereits mit C vertraut sind, können Sie die folgenden Zeilen als »Update« betrachten, das Sie darüber informiert, was mit C++ alles neu auf Sie (als C-Umsteiger) zukommt.

Erweiterungen von C

Neben der objektorientierten Unterstützung bietet C++ auch sprachliche Erweiterungen von C an. Zu den bekanntesten Erweiterungen gehören folgende Punkte:

- Inline-Funktionen
- Default-Argumente
- Referenztypen
- Überladen von Funktionen
- Überladen von Operatoren
- Templates
- Ausnahmebehandlung

Objektorientierte Unterstützung

Die objektorientierte Unterstützung von C++ enthält folgende Möglichkeiten:

- Klassen bilden
- Zugriff auf Daten und Elementfunktionen mit `public`-, `private`- oder `protected`-Spezifikationen steuern
- Klassen vererben (auch mehrfach)
- polymorphe Klassen bilden

Stärken von C++

Wie bereits erwähnt, hat C++, wie jede andere Sprache auch, einige »schwache« und »starke« Seiten. Zu den Stärken von C++ gehören folgende Punkte:

- maschinennahes Programmieren
- Erzeugung von hocheffizientem Code
- hohe Ausdrucksstärke und Flexibilität
- für umfangreiche Projekte geeignet
- sehr weite Verbreitung
- Keine Organisation (wie z. B. bei Java) hat hier ihre Finger mit im Spiel (die Standardisierung erfolgt durch die ISO).
- viele Möglichkeiten für die Metaprogrammierung
- C-kompatibel – dadurch kann das Programm, das in C erstellt wurde, weiterhin unverändert verwendet werden. Außerdem braucht sich ein C-Programmierer beim Umstieg nur mit den Erweiterungen und der objektorientierten Programmierung auseinanderzusetzen.

Schwächen von C++

Die »Schwächen« von C++, die hier erwähnt werden, sind allerdings häufig auch Schwächen, die viele andere Programmiersprachen auch aufweisen.

- C-kompatibel – Wie schon erwähnt, hat die Kompatibilität zu C zwar Vorteile, aber leider muss C++ auch den Balast von C mitschleppen. Dies hat den Nachteil, dass einige Details der Sprache compilerspezifisch sind, obwohl sie es nicht sein müssten. Dies erschwert die Portierung von C++-Programmen auf die verschiedenen Rechnertypen, Betriebssysteme und Compiler.
- Kaum ein Compiler erfüllt die komplette Umsetzung der ISO-Norm (ein ähnliches Problem gibt es auch mit C-Compilern und dem »C99-Standard«).
- C++ gilt als relativ schwierig zu erlernen – es ist eine längere Einarbeitungszeit nötig.
- Es gibt (noch) keine Standardbibliotheken zu vielbenutzten Themen wie Multithreads, Socket-Programmierung und den Dateisystem-Verzeichnissen. Dies erschwert die Portabilität von C++ nochmals erheblich. Zwar wird hierbei auf viele externe Bibliotheken zurückgegriffen, doch dies vermittelt wohl eher ein uneinheitliches und nicht ausgereiftes Bild von C++. Es existiert schon seit Längerem eine Sammlung von Bibliotheken, die unter dem Namen *Boost* zusammengefasst werden, aber diese Bibliotheken sind (noch) kein Standard.

Vergleich mit anderen Sprachen

Der Vergleich verschiedener Programmiersprachen ist häufig unsinnig, da jede Programmiersprache generell als Mittel zum Zweck dient und somit auch ihre Vor- bzw. Nachteile hat. Somit will ich C++ nicht im Vergleich zu anderen Sprachen auf Vor- bzw. Nachteile prüfen, sondern vielmehr auf Ähnlichkeiten.

Die Programmiersprachen Java und C# (gesprochen »C Sharp«) haben zum Beispiel eine ähnliche Syntax wie C++. Allerdings sind beide Sprachen »intern« komplett anders aufgebaut und somit auch gänzlich inkompatibel.

Ganz unumstritten ist immer noch, dass C++ (mit seinem Ursprung C) eine der am häufigsten eingesetzten und geforderten Programmiersprachen ist. In vielen Bereichen kann man allerdings mit C# oder Java besser (bzw. auch schneller) ans Ziel kommen. Würde C++ nicht die generische Programmierung beherrschen, wäre die Sprache wohl stark rückläufig.

1.2 Erste Schritte der C++-Programmierung

Zur Entwicklung eines C++-Programms sind im Grunde nur drei Dinge nötig: ein Editor, mit dem Sie die Textdatei mit dem Code erstellen, ein Compiler, der das Programm in die Maschinensprache des entsprechenden Rechners übersetzt, und ein Linker, der daraus eine ausführbare Datei macht. Gewöhnlich werden die Quelldateien mit der Endung *.cpp* oder *.cc* gespeichert – auch mit *.C* (großes C) und *.cxx* kommt g++ zurecht. Die Header-Dateien erhalten entweder die Endung *.h*, *.hpp* oder überhaupt keine Endung.

Hinweis [«]

Da Textverarbeitungsprogramme wie MS Word oder WordPerfect zusätzlichen Formatierungsbalast beim Speichern hinzufügen, eignen sich diese weniger zur Erstellung der Quelldatei. Sie benötigen auf jeden Fall einen Texteditor, mit dem Sie ASCII-Dateien editieren können.

Mit dem Compiler erzeugen Sie aus einer Quelldatei und den inkludierten Header-Dateien eine Objektdatei (auch Modul genannt), die den Maschinencode enthält. Natürlich ist dieser Maschinencode nicht mehr portabel.

Maschinensprache (Maschinencode)

Maschinensprache nennt man den Befehlssatz eines Mikroprozessors, gleichsam seine Muttersprache. Im Gegensatz zur maschinennahen Assemblersprache oder Hochsprachen (wie bspw. C/C++ eine ist) handelt es sich um eine für den Menschen kaum lesbare Sprache, die allenfalls von Experten mit einem sogenannten Maschinensprachemonitor bearbeitet werden kann.

Der Linker bindet anschließend diese Objektdatei(en) zu einer ausführbaren Datei . Diese Datei enthält neben den erzeugten Objektdateien auch den Startup-Code und die Module mit den verwendeten Funktionen und Klassen der Standardbibliothek (siehe Abbildung 1.1).

Häufig wird zur Erzeugung eines Programms auch auf *IDEs (Entwicklungsumgebungen)* zurückgegriffen. Eine IDE enthält schlicht und einfach alle Werkzeuge wie Editor, Compiler und Linker in »einem Fenster«. Neben den üblichen Werkzeugen zur Entwicklung umfassen solche Entwicklungsumgebungen noch eine ganze Menge mehr, wie zum Beispiel einen Debugger, einen Profiler oder einen Hexeditor.

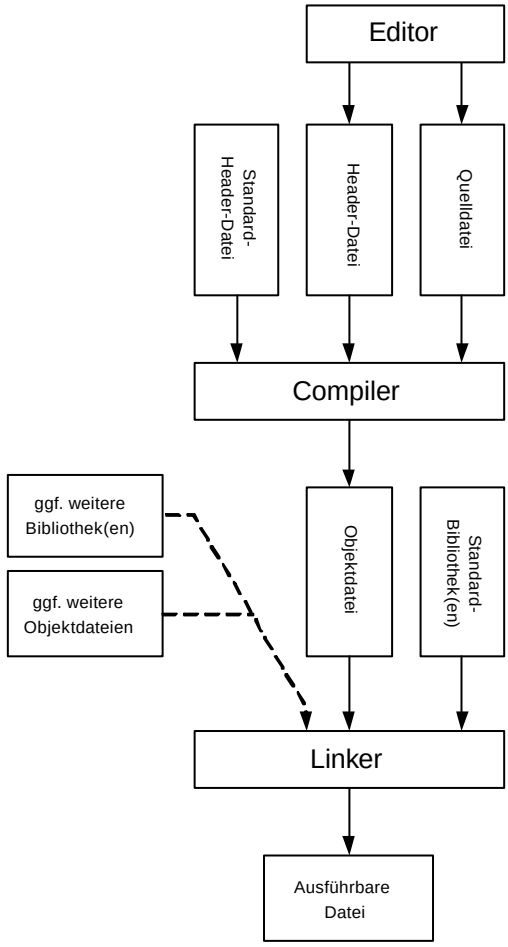

Abbildung 1.1 Vom Quellcode zur ausführbaren Datei

1.2.1 Ein Programm erzeugen mit einem Kommandozeilen-Compiler

Als Buchautor steht man immer vor der Frage, welchen Compiler man hier mit aufnehmen bzw. beschreiben soll. Und zu der Frage des Compilers kommt auch noch das Betriebssystem dazu – also kein leichtes Unterfangen, jeden Leser zufriedenzustellen. Zum Glück macht es C++ mit einem Standard hier recht leicht, so dass die Beispiele im Buch nicht vom System oder Compiler abhängen. Und wenn etwas eingesetzt wird, was nicht dem Standard entspricht, so finden Sie auch in diesem Buch für jedes System eine Lösung, um das Programm dennoch auszuführen. Darüber brauchen Sie sich jedoch an dieser Stelle noch nicht den Kopf zu zerbrechen.

Zunächst müssen Sie den Quelltext in einen ASCII-Editor eintippen und mit der Endung *.cpp* oder *.cc* speichern. Natürlich werden wir hierzu das klassische »Hallo Welt«-Programm verwenden.

```
// hallo.cpp
#include <iostream>
using namespace std;

int main(void) {
    cout << "Hallo Welt!" << endl;
    return 0;
}
```

Ungeachtet dessen, was diese Zeilen bedeuten, soll hieraus ein ausführbares Programm erstellt werden. Hierzu müssen Sie lediglich den Compiler (und Linker) aufrufen.

Hinweis **[«]**

Was ist mit dem Linkerlauf, mag sich manch einer fragen. Dieser wird in der Praxis automatisch mit ausgeführt, sofern dies nicht explizit anders gewollt (genauer: über Compiler-Flags angegeben) ist.

GNU g++-Compiler

Der GNU g++-Compiler ist Bestandteil der freien *GNU Compiler Collection* (kurz *GCC*). Häufig wird dieser Compiler auch in integrierten Entwicklungsumgebungen wie zum Beispiel unter Linux in KDevelop und Anjuta oder unter Windows mit Dev-C++ ausgeführt. Unter Windows ist der g++ auch als MinGW-Compiler bekannt. Neben Linux und Windows ist g++ auch unter UNIX und Mac OS X (kostenlos unter *http://www.gnu.org/*) erhältlich. Folgendermaßen können Sie *hallo.cpp* mit g++ in der Kommandozeile übersetzen:

```
$ g++ -Wall -o hallo hallo.cpp
```

Mit dem Schalter `-o` (output) teilen Sie dem Compiler mit, dass die ausführbare Datei den Namen *hallo* haben soll. Mit `-Wall` (Warnings all) schalten Sie alle Warnungen ein, wodurch der Compiler viele Extrahinweise mit ausgibt. *hallo.cpp* lautet der Name der Quelldatei, die in eine ausführbare Datei übersetzt werden soll.

Microsoft Visual C++ (Kommandozeile)

Der Microsoft Visual C++-Kommandozeilen-Compiler `cl` ist Teil der Visual Studio-Entwicklungsumgebung – er ist aber auch bei Microsoft kostenlos erhältlich (besser bekannt unter »Visual C++ 2003 Toolkit«). Allerdings enthält die kosten-

lose Version des Microsoft-Compilers keine IDE. *hallo.cpp* übersetzen Sie also mit dem Microsoft-Compiler wie folgt:

```
C:> cl /Fe hallo /Wall hallo.cpp
```

Mit `/Fe` – File(name) executable; Name der EXE-Datei – teilen Sie dem Compiler mit, dass Sie das ausführbare Programm *hallo.exe* nennen wollen. Mit `/Wall` (Warnings all) schalten Sie die Warnmeldungen ein (optional). Auch hier gilt, *hallo.cpp* ist der Name der Quelldatei, aus der eine ausführbare Datei erstellt werden soll.

Borlands Free Command Line Tools

Der Borland C++-Kommandozeilen-Compiler ist ebenfalls kostenlos für Windows erhältlich. Er ist ein freigegebener Bestandteil von Borlands C++-Compiler, nur ohne grafische Oberfläche. Der Compiler ist bei vielen C++-Anfängern sehr beliebt, so dass es mittlerweile einen Installer sowie zahlreiche IDEs (bspw. VIDE) dafür gibt. Das »Hallo Welt«-Programm übersetzen Sie mit Borlands Kommandozeilen-Compiler wie folgt:

```
C:> bcc32 -w -tWC -e hallo hallo.cpp
```

Die Warnmeldungen schalten Sie mit der Option `-w` ein. Mit `-tWC` teilt man dem Compiler mit, dass man eine Konsolenapplikation erzeugen will. Das ist ein Programm, das die Standard-C++-API und ein MS-DOS-Fenster für die Ein- und Ausgabe verwendet. Mit `-e` geben Sie an, dass ein Programm mit dem Namen *hallo.exe* erzeugt werden soll. *hallo.cpp* ist der Name der Quelldatei.

1.2.2 Ausführen des Programms

Ausführen können Sie das Programm mit einem Namen, den Sie in der Kommandozeile bei der entsprechenden Option angegeben haben. Im Beispiel wurde hier immer der Name »hallo« verwendet. Somit genügt ein einfaches »hallo« (gegebenenfalls auch »hallo.exe«) bzw. unter Linux/UNIX »./hallo« in der Kommandozeile, um das Programm zu starten. Anschließend wird die Meldung `"Hallo Welt!"` auf dem Bildschirm ausgegeben. Natürlich wird davon ausgegangen, dass Sie sich im Augenblick im aktuellen Verzeichnis befinden, in dem das Programm enthalten ist.

1.2.3 Ein Programm erzeugen mit einer IDE

Die Erstellung und Übersetzung ist natürlich mit einer integrierten Entwicklungsumgebung wesentlich komfortabler und häufig mit einigen Mausklicks schneller erledigt. Schließlich findet man alles, was man benötigt, an einem Fleck. Aller-

dings gibt es mittlerweile eine Menge interessanter IDEs, so dass an dieser Stelle nicht darauf eingegangen wird.

Hinweis [«]

Wenn Sie ein einfaches C++-Programm mit so großen Entwicklungsumgebungen wie beispielsweise MS Visual C++/C# übersetzen wollen, ist das wie mit Kanonen auf Spatzen schießen. Immer noch sind viele Anfänger verwundert, wenn ich ihnen sage, dass sie theoretisch alles ohne eine solche Entwicklungsumgebung programmieren können. Lassen Sie sich nicht verunsichern, wenn jemand behauptet, Sie benötigten diese oder jene Entwicklungsumgebung, um Programme zu erstellen. Entwicklungsumgebungen können einem das Leben erheblich erleichtern, einem Anfänger dagegen kann solch eine Software schnell das (Programmierer-)Leben vermiesen.

Hinweis [«]

Da viele Leser ihre Programme gerne mit einer Entwicklungsumgebung erstellen wollen, finden Sie auf der Buch-CD kurze Einführungen zu den gängigsten Entwicklungsumgebungen.

1.3 Symbole von C++

Wenn man sich mit einer Sprache befassen muss/will, sollte man sich auch mit den gültigen Symbolen auseinandersetzen.

1.3.1 Bezeichner

Den Begriff *Bezeichner* verwendet man für Namen von Objekten im Programm. Dazu gehören Variablen, Funktionen, Klassen usw.

Ein gültiger Bezeichner darf aus beliebigen Buchstaben, Ziffern und dem Zeichen _ (Unterstrich) bestehen. Allerdings darf das erste Zeichen niemals eine Ziffer sein. Man sollte außerdem beachten, dass C++ zwischen Groß- und Kleinbuchstaben (engl.: *case sensitiv*) unterscheidet. Somit sind `Hallo`, `hallo` und `HALLO` drei verschiedene Bezeichner.

1.3.2 Schlüsselwörter

Schlüsselwörter sind Bezeichner mit einer vorgegebenen Bedeutung in C++ und dürfen nicht anderweitig verwendet werden. So dürfen Sie zum Beispiel keine Variable mit dem Bezeichner `int` verwenden, da es auch einen Basisdatentyp hierzu gibt. Der Compiler würde sich ohnehin darüber beschweren. Eine Liste der Schlüsselwörter in C++ finden Sie im Anhang des Buches.

1.3.3 Literale

Als Literale werden Zahlen, Zeichenketten und Wahrheitswerte im Quelltext bezeichnet, die ebenfalls nach einem bestimmten Muster aufgebaut sein müssen, das heißt, Literale sind von einer Programmiersprache definierte Zeichenfolgen zur Darstellung der Werte von Basistypen.

Ganzzahlen

Man unterscheidet bei Ganzzahlen zwischen Dezimal-, Oktal- und Hexadezimalzahlen, wofür folgende Regeln gelten:

- Dezimalzahlen (Basis 10) – Eine Dezimalzahl besteht aus einer beliebig langen Ziffernreihe aus den Zeichen 0 bis 9. Die erste Ziffer darf allerdings keine 0 sein.
- Oktalzahlen (Basis 8) – Eine Oktalzahl hingegen beginnt immer mit einer 0, gefolgt von einer Reihe von Oktalzahlen (0–7).
- Hexadezimalzahlen (Basis 16) – Eine Hexadezimalzahl beginnt immer mit der Sequenz 0x bzw. 0X gefolgt von einer Reihe von Hexadezimalzahlen (0–F = 0 1 2 3 4 5 6 7 8 9 A B C D E F – oder Kleinbuchstaben: a b c d e f).

[»] **Hinweis**

Mehr zum jeweiligen Zahlensystem entnehmen Sie bitte dem Anhang dieses Buches.

Man kann an die Dezimal-, Oktal- und Hexadezimalzahlen noch ein Suffix anhängen, mit dem der Wertebereich einer Zahl genauer spezifiziert werden kann. Das Suffix `u` bzw. `U` deutet zum Beispiel an, dass es sich um eine vorzeichenlose (`unsigned`) Zahl handelt. `l` bzw. `L` gibt an, dass es sich um eine `long`-Zahl handelt. Hierzu einige Beispiele, die in der Reihe immer gleichwertig sind:

Dezimalzahl	Oktalzahl	Hexadezimalzahl
123	0173	0x7B
1234567L	04553207L	0X12D687L
66u	0102U	0x42u

Tabelle 1.1 Beispiele für gültige Ganzzahlen

Fließkommazahlen

Wie eine korrekte Fließkommazahl dargestellt wird, wird in Abschnitt 1.4.6, »Gleitkommazahlen ›float‹, ›double‹ und ›long double‹«, genauer beschrieben, wenn es um die Basistypen von Fließkommazahlen geht. Wie bei den Ganzzahlen kann man den Fließkommazahlen ebenfalls ein Suffix hinzufügen. Mit dem

Suffix `f` oder `F` kennzeichnet man eine Fließkommazahl mit einer einfachen Genauigkeit. Das Suffix `l` oder `L` hingegen deutet auf eine Fließkommazahl mit erhöhter Genauigkeit hin.

Einzelne Zeichen

Ein Zeichen-Literal wird in einfache Hochkommata (*Single Quotes*) eingeschlossen (`'A'`, `'B'`, `'C'`, ... `'$'`, `'&'` usw.). Will man nicht druckbare Zeichen wie beispielsweise einen Tabulator oder einen Zeilenvorschub darstellen, muss man eine Escape-Sequenz (auch Steuerzeichen genannt) verwenden. Escape-Sequenzen werden mit einem Backslash (\) eingeleitet (bspw. Tabulator = `'\t'` oder neue Zeile = `'\n'`). Mehr zu den Escape-Sequenzen erfahren Sie in Abschnitt 1.4.4, »Der Datentyp ›char‹«.

Zeichenketten

Eine Zeichenkette ist eine Sequenz von Zeichen, die zwischen doppelte Hochkommata (*Double Quotes*) gestellt wird (bspw. `"Ich bin eine Zeichenkette"`). Sehr wichtig im Zusammenhang mit einer Zeichenkette ist es zu wissen, dass jede dieser Ketten um ein Zeichen länger ist als (sichtbar) dargestellt. Gewöhnlich werden Zeichenketten durch das Zeichen mit dem ASCII-Wert 0 (nicht die dezimale 0) abgeschlossen (0x00 oder als einzelnes Zeichen `'\0'`). Diese ASCII-0 kennzeichnet immer das Ende einer Zeichenkette. Somit enthält zum Beispiel die Zeichenkette `"C++"` vier Zeichen, weil am Ende auch das Zeichen 0x00 (oder auch `'\0'`) abgelegt ist.

1.3.4 Einfache Begrenzer

Um die Symbole voneinander zu trennen, benötigt man auch Begrenzer in C++. In diesem Abschnitt wird nur auf einfache Begrenzer hingewiesen. Weitere Begrenzer werden Sie im Verlaufe des Buches näher kennenlernen.

Das Semikolon

Der wichtigste Begrenzer dürfte das Semikolon ; sein (Plural: Semikola und Semikolons, auch Strichpunkt genannt). Dieser dient als Abschluss einer Anweisung. Jeder Ausdruck, der mit einem solchen Semikolon endet, wird als Anweisung behandelt. Der Compiler weiß dann, dass hier das Ende der Anweisung ist, und fährt nach der Abarbeitung der Anweisung (des Befehls) mit der nächsten Zeile bzw. Anweisung fort. Natürlich hat das Semikolon keine Wirkung, wenn es in einer Stringkonstante verwendet wird:

```
"Hallo; Welt"
```

Komma

Mit dem Komma trennt man gewöhnlich die Argumente einer Funktionsparameterliste oder bei der Deklaration mehrere Variablen desselben Typs.

Geschweifte Klammern

Zwischen den geschweiften Klammern – *braces* (amerikanisch) oder *curly brackets* (britisch) – wird der Anweisungsblock zusammengefasst. In diesem Block befinden sich alle Anweisungen (abgeschlossen mit einem Semikolon), die in einer Funktion ausgeführt werden sollen. Bei dem Listing *hallo.cpp*

```
int main(void) {
    cout << "Hallo Welt!";
    return 0;
}
```

finden Sie alle Anweisungen der `main()`-Funktion zwischen den geschweiften Klammern zusammengefasst. Hier wird lediglich die Textfolge `"Hallo Welt!"` auf dem Bildschirm ausgegeben und mit `return` der Wert 0 an den aufrufenden Prozess zurückgegeben. Mehr zur `main()`-Funktion und dessen Rückgabewert erfahren Sie etwas später.

Das Gleichheitszeichen

Mit dem Gleichheitszeichen trennt man die Variablendeklaration von den Initialisierungslisten oder bei Parameterlisten einer Funktion den Vorgabewert eines Parameters.

```
Typ name = wert;
// oder als Vorgabewert einer Funktion
typ function ( typ name = wert );
```

[»] **Hinweis**

Wenn Sie eine Variable initialisieren, so ist dies noch lange keine Zuweisung. Ist das nicht ein und dasselbe? Nicht ganz. Zwar erfolgen die Zuweisung und die Initialisierung mit dem Gleichheitszeichen, aber zum einen handelt es sich ja hier um den Begrenzer =, und zum anderen um den Operator =, also um zwei verschiedene Dinge, die mit ein und demselben Zeichen einhergehen. Eine Initialisierung geschieht immer erst beim Anlegen einer Variablen, während eine Zuweisung lediglich in Bezug auf ein bereits existierendes Objekt ausgeführt werden kann.

[»] **Hinweis**

Wenn Sie in einem Programm auf zwei aufeinanderfolgende == stoßen, so handelt es sich hierbei nicht mehr um eine Zuweisung, sondern um eine Prüfung.

1.4 Basisdatentypen

Als Basisdatentypen werden einfache vordefinierte Datentypen bezeichnet. Dies umfasst in der Regel den Wahrheitswert (`bool`), die Zahlen (`int`, `short int`, `long int`, `float`, `double` und `long double`), die Zeichen (`char`, `wchar_t`) und den (Nichts-)Typ (`void`).

1.4.1 Deklaration und Definition

Den etwas unbequemeren Abschnitt zuerst: Deklaration und Definition werden oft durcheinandergebracht oder auch als ein und dasselbe verwendet. Mit einer Deklaration machen Sie den Compiler mit einem Namen (Bezeichner) bekannt und verknüpfen diesen Namen mit einem Typ. Der Typ wiederum enthält die Informationen über die Art der Bezeichner und bestimmt somit implizit die Aktionen, die auf das Speicherobjekt zulässig sind. Bei einer Ganzzahl zum Beispiel sind die arithmetischen Operationen `+`, `-`, `*`, `/` usw. als Aktionen zulässig. Die Syntax einer einfachen Deklaration sieht somit immer wie folgt aus:

```
Typ name;
Typ name1, name2, name3;
```

Mit `Typ` geben Sie immer den Datentyp an, und `name` ist immer der Bezeichner. Natürlich können Sie, wie im zweiten Beispiel gesehen, auch mehrere Bezeichner eines Typs durch Kommata voneinander trennen.

Mit einer Deklaration geben Sie dem Compiler nur Informationen zum Typ bekannt. Bis dahin wurde noch keine Zeile Maschinencode erzeugt, geschweige denn ein Speicherobjekt (Variable) angelegt.

Für das konkrete Speicherobjekt im Programm bzw. den ausführbaren Code wird die Definition vereinbart. Somit ist jede Definition gleichzeitig auch eine Deklaration. Gleiches gilt auch häufig andersherum. Die Deklaration einer Variablen vom Datentyp `long` in der Art

```
long i;
```

ausgeführt, gibt zum Beispiel den Namen des Speicherobjekts bekannt und vereinbart somit auch den Speicherplatz für das Objekt. Ebenso kann der Name einer Variablen vereinbart werden, ohne dass ein Objekt erzeugt wird (Speicherklassenattribut `extern`). Damit kann es für jedes Objekt im Programm zwar beliebig viele Deklarationen geben, aber nur eine einzige Definition.

[»] **Hinweis**

Im Gegensatz zu C können in C++ Deklarationen überall im Quelltext vorgenommen werden – worauf man allerdings, wenn möglich, wegen der Lesbarkeit des Codes verzichten sollte, indem man die Deklarationen noch vor den »arbeitenden« Anweisungen setzt.

1.4.2 Was ist eine Variable?

Eine Variable ist eine Stelle (Adresse) im Hauptspeicher (RAM), an der Sie einen Wert ablegen und gegebenenfalls später wieder darauf zurückgreifen können. Neben einer Adresse hat eine Variable auch einen Namen, genauer einen Bezeichner, mit dem man auf diesen Wert namentlich zugreifen kann. Und natürlich belegt eine Variable auch eine gewisse Größe des Hauptspeichers, was man mit dem Typ der Variablen mitteilt. Rein syntaktisch kann man das wie folgt betrachten:

```
long lvar;
```

Hier haben Sie eine Variable mit dem Namen (Bezeichner) `lvar` vom Typ `long`, der üblicherweise vier Bytes (auf 32-Bit-Systemen) im Hauptspeicher (RAM) belegt. Wo (Speicheradresse) im Arbeitsspeicher Speicherplatz für diese Variable reserviert wird – hier vier Bytes –, können Sie nicht beeinflussen.

1.4.3 Der Datentyp »bool«

Mit dem Datentyp `bool` können Sie Wahrheitswerte beschreiben. Dabei kann `bool` die Werte `true` (für »wahr«) und `false` (für »falsch«) ausdrücken.

```
bool flag;
...
flag = true;  // Schalter auf wahr setzen
...
flag = false; // Schalter auf falsch setzen
```

Der Sinn dieses Datentyps wird genauer erläutert, wenn es um Logikanweisungen geht.

[»] **Hinweis**

Auch in C – wenn auch erst im C99-Standard (ISO/IEC 9899:1999) – gibt es mittlerweile einen Datentyp `bool`, nur dass dieser hier `_Bool` lautet.

Hinweis [«]

Boolesche Variablen sind nach dem englischen Mathematiker George Boole benannt, der mit seiner Theorie der booleschen Algebra einen Grundstein für die formale Logik und die Rechentechnik legte.

1.4.4 Der Datentyp »char«

Der grundlegende Datentyp für Zeichen lautet `char` und belegt gewöhnlich ein Byte an Speicherplatz, was somit meistens $2^8 = 256$ Ausprägungen entspricht. Allerdings muss ein Byte nicht zwangsläufig aus acht Bits bestehen. Es gab früher auch Maschinen, die zum Beispiel neun Bit als kleinsten adressierbaren Typ hatten. Des Weiteren gibt es zum Beispiel DSPs, bei denen ein Byte 32 Bit entspricht. Damit kann ein `char` auch von -2^{31} bis $+ 2^{31-1}$ gehen.

DSP

DSPs sind digitale Signalprozessoren und haben eine für rechenintensive Anwendungen optimierte Architektur. Sie können komplexe Algorithmen, wie sie bei der Verarbeitung digitalisierter Analogsignale notwendig sind, sehr schnell abarbeiten. Anwendungen ergeben sich in der Signalfilterung, in der Signalkodierung und -dekodierung, aber auch in der Bildverarbeitung.

Hinweis [«]

Wie viele Bits ein `char` auf Ihrem System denn nun hat, ist im Makro `CHAR_BIT` (in der Header-Datei <*climits*> alias <*limits.h*>) definiert. Aber egal, wie viele Bits ein `char` dabei hat, ein `sizeof(char)` muss immer eins (ein Byte) ergeben!

Im Zeichentyp `char` findet der komplette ASCII-Zeichensatz Platz. Da die ASCII-Zeichen geordnet sind, kann man diese auch miteinander vergleichen. Natürlich kann man hierbei auch Umwandlungen von Zeichen in Zahlen vornehmen. Beispielsweise entspricht das Zeichen `'A'` (laut ASCII-Tabelle) dem Zahlenwert 65, `'B'` entspricht 66 usw.

Hinweis [«]

Den Zeichentyp `char` kann man zwar auch mit `signed` oder `unsigned` spezifizieren, aber man sollte beachten, dass `char`, `unsigned char` und `signed char` drei verschiedene Typen sind! Des Weiteren hängt es von der Compiler-Implementierung ab, ob `char` auch negative Zahlen aufnehmen kann.

Zeichen werden bei `char` in einfachen Anführungsstrichen eingeschlossen:

```
char ch1 = 'A';
char ch2 = 'B';
```

Hierbei handelt es sich immer um Zeichenkonstanten. Natürlich können Sie auch, wie schon erwähnt, den entsprechenden ASCII-Zahlenwert verwenden. So entspricht die eben gezeigte Syntax den folgenden Zahlenwerten:

```
char ch1 = 65;  // laut ASCII-Tabelle das Zeichen 'A'
char ch2 = 66;  // laut ASCII-Tabelle das Zeichen 'B'
```

Hierzu ein einfaches Beispiel:

```
// char1.cpp
#include <iostream>
using namespace std;

int main() {
    char ch1 = 'A';
    char ch2 = 'B';
    char ch3 = 67;  //laut ASCII-Tabelle 'C'
    char ch4 = 68;  //laut ASCII-Tabelle 'D'
    cout << ch1 << ch2 << ch3 << ch4 << '\n';
    return 0;
}
```

Die Ausgabe des Programms lautet:

```
ABCD
```

Neben den darstellbaren Zeichen können Sie auch Sonderzeichen (nicht druckbare Zeichen) bzw. Escape-Zeichen (oder auch Fluchtzeichen) verwenden, die mit einem Backslash (\) eingeleitet, aber als einzelnes Zeichen interpretiert werden. So erreichen Sie mit dem Escape-Zeichen `'\n'`, dass ein Zeilenvorschub »ausgegeben« wird, mit `'\t'` wird das Tabulator-Zeichen ausgegeben – meistens wird dabei in der Zeile um acht Zeichen eingerückt (abhängig vom System bzw. der Shell-Einstellung). Hierzu ein Überblick über die nicht druckbaren Steuerzeichen und deren Bedeutung:

Steuerzeichen	Bedeutung
`\a`	BEL (*bell*) – akustisches Warnsignal
`\b`	BS (*backspace*) – setzt den Cursor um eine Position nach links.
`\f`	FF (*formfeed*) – ein Seitenvorschub wird ausgelöst. Wird hauptsächlich bei Programmen verwendet, mit denen Sie etwas ausdrucken können.
`\n`	NL (*newline*) – Cursor geht zum Anfang der nächsten Zeile.
`\r`	CR (*carriage return*) – Cursor springt zum Anfang der aktuellen Zeile.
`\t`	HT (*horizontal tab*) – Zeilenvorschub zur nächsten horizontalen Tabulatorposition (meistens acht Leerzeichen weiter)

Tabelle 1.2 Steuerzeichen (Escape-Sequenzen) in Zeichenkonstanten

Steuerzeichen	Bedeutung
\v	VT (*vertical tab*) – Cursor springt zur nächsten vertikalen Tabulatorposition.
\"	" wird ausgegeben.
\'	' wird ausgegeben.
\?	? wird ausgegeben.
\\	\ wird ausgegeben.
\0	Ist die Endmarkierung eines Strings.
\nnn	Ausgabe eines Oktalwerts (z. B. \033 = Escape-Zeichen)
\xhh	Ausgabe eines Hexadezimalwerts

Tabelle 1.2 Steuerzeichen (Escape-Sequenzen) in Zeichenkonstanten (Forts.)

Hierzu ebenfalls ein recht einfaches Beispiel:

```
// char2.cpp
#include <iostream>
using namespace std;

int main(void) {
    char ch1 = 'A';
    char ch2 = 'B';
    char ch3 = '\t';
    char ch4 = '\n';
    cout << ch1 << ch3 << ch2 << ch4 << ch1 << ch2 << ch4;
    return 0;
}
```

Die Ausgabe des Programms lautet:

```
A       B
AB
```

Hinweis [«]

Beachten Sie bitte, dass einzelne Zeichen bei C++ in einfache Anführungsstriche eingeschlossen werden, Zeichenketten (Strings) hingegen von doppelten Anführungsstrichen umschlossen werden.

Breite Zeichen – »wchar_t«

char reicht für die englischen und westeuropäischen Sprachen (oder genauer Zeichensätze) zwar aus, aber für Sprachen, die mehr Zeichen haben, als man zu zählen vermag (z. B. Chinesisch mit mehreren tausend Zeichen), benötigt man den Datentyp wchar_t, mit dem »breite« Zeichen ausgegeben werden können. Aller-

dings wird bei der Deklaration von breiten Zeichen noch vor dem Anführungszeichen das Präfix `L` verwendet:

```
wchar_t omega = L'Z';    // griechisches 'Z' (Omega)
```

Zeichensatz

Zeichensatz nennt man die Zuordnung der alphanumerischen Zeichen zu einer Zahl. Die wichtigsten in der Informatik bekannten Zeichencodierungen sind der ASCII- und der EBCDIC-Code; insbesondere Letzterer hat aber stark an Bedeutung verloren. Zunehmend in den Vordergrund getreten sind Zeichensätze mit international notwendigen Zeichen, die über das Englische hinausgehen, zum Beispiel diesbezügliche Zeichensätze gemäß ANSI, vor allem der international anerkannte Standard *Unicode*.

1.4.5 Die Datentypen »int«

Der Datentyp für Ganzzahlen lautet `int`. Ein `int` hat somit laut Standard die natürliche Größe, die von der Ausführumgebung (engl.: *Execution-Environment*) vorgeschlagen wird, oder einfach die Größe eines Maschinenworts des entsprechenden Rechnersystems. Das wären dann zum Beispiel auf einer PDP10-Maschine 36 Bit, auf einem Pentium 4 32 Bit und auf einem beliebigen 64-Bit-Prozessor-System 64 Bit.

PDP

PDP ist eine Rechnerfamilie der Firma DEC. PDP steht als Abkürzung für *Programmable Data Processor* oder *Programmed Data Processor*.

Auf 16-Bit-Systemen lässt sich hiermit in den zwei Bytes (2^{16}) ein Zahlenbereich von –32768 bis +32767 beschreiben. Auf 32-Bit-Rechnern sind dies schon vier Bytes, mit denen man einen Zahlenraum von (2^{32}) –2147483648 bis +2147483647 verwenden kann. In der aktuellen 64-Bit-Generation belegt ein `int` schon beachtliche acht Bytes, mit denen ein Zahlenbereich von –9.223.372.036.854.755.808 bis +9.223.372.036.854.755.807 dargestellt werden kann.

[»] **Hinweis**

Für 32-Bit-Rechner gibt es den Datentyp `long long`, mit dem sich ebenfalls dieser 64 Bit (acht Byte) breite Wert beschreiben lässt.

[»] **Hinweis**

Die Konstanten `INT_MIN` und `INT_MAX` der Header-Datei <*climits*> (in C auch bekannt als <*limits.h*>) sind mit den Werten deklariert, die der Datentyp `int` auf Ihrem System besitzt.

Hierzu ein einfaches Beispiel:

```
// int1.cpp
#include <iostream>
#include <climits>
using namespace std;

int main(void) {
    int wert1 = 10;
    int wert2 = 20;
    cout << "wert1: " << wert1 << '\n';
    cout << "wert2: " << wert2 << '\n';
    cout << "int-Zahlenbereich von "
         << INT_MIN << " bis " << INT_MAX << '\n';
    return 0;
}
```

Die Ausgabe des Programms lautet (hier auf einem Pentium 4 – 32 Bit):

```
wert1: 10
wert2: 20
int-Zahlenbereich von -2147483648 bis +2147483647
```

Die int-Typen »long« und »short«

Ebenfalls zur Familie der `int`-Typen gehören die Datentypen `short` und `long`.

Hinweis **[«]**

Um das hier gleich richtigzustellen, die korrekten Typnamen sind eigentlich `short int` und `long int`. In der Praxis werden üblicherweise `short` und `long` verwendet.

Wie man am Namen schon ablesen kann, handelt es sich um eine kurze (`short`) und eine lange (`long`) Version von `int`.

Hinweis **[«]**

Die Größe der Typen `int`, `short` und `long` ist nicht festgelegt. `int` weist die Größe eines Maschinenworts auf und ist mindestens so groß wie `short`. Der Datentyp `long` hingegen hat mindestens die Ausdehnung eines `int`.

Der Datentyp `short` hat somit mindestens den Wertebereich von `char` und maximal den von einem `int`. Gewöhnlich kann man mit `short` (2 Bytes) einen Wertebereich von –32768 bis +32767 (oder 0 bis 65535) beschreiben (2^{16}).

Der Datentyp `long` hingegen wird gewöhnlich verwendet, wenn der Zahlenbereich von `int` zu gering ist. `long` deckt mindestens den Bereich eines `int` ab. Auf

16- und 32-Bit-Rechnern kann man mit `long` somit immer einen Zahlenbereich von –2147483648 bis +2147483647 (bzw. 0 bis 4294967296) beschreiben (2^{32}).

Natürlich soll hier nicht der Datentyp `long long` vergessen werden. Bei den neueren C++-Compilern ist dieser Datentyp gewöhnlich schon integriert und deckt mit seinen acht Bytes Breite einen Zahlenbereich von –9.223.372.036.854.755.807 bis +9.223.372.036.854.755.807 ab (2^{64}).

»signed« und »unsigned«

Jeden dieser `int`-Typen können Sie noch zusätzlich mit `signed` bzw. `unsigned` spezifizieren. Zwar weisen beide Spezifizierungen dieselbe Ausdehnung auf, aber die Typen verfügen über einen anderen Wertebereich. Abgesehen vielleicht von `char` (compilerabhängig) ist `signed` die Voreinstellung der Datentypen, mit der auf eine explizite Angabe verzichtet werden könnte. Folgende Schreibweise hätte somit dieselbe Bedeutung, wobei die zweite Variante die Lesbarkeit eines Quelltexts erhöhen kann:

```
int a;         // gleich wie: signed int a
signed int a;  // gleich wie: int a
```

Hierzu ein Überblick, wie sich das Schlüsselwort `signed` bzw. `unsigned` auf den Wertebereich eines 32-Bit-Systems auswirkt:

Name	Wertebereich
`char`, `signed char`	–128…+127 bzw. 0…255 (compilerabhängig)
`unsigned char`	0…255
`short`, `signed short`	–32768…+32767
`unsigned short`	0…65535
`int`, `signed int`	–2147483648…+2147483648
`unsigned int`	0…4294967295
`long`, `signed long`	–2147483648…+2147483648
`unsigned long`	0…4294967295

Tabelle 1.3 Wertebereich in Verbindung mit »signed« und »unsigned«

1.4.6 Gleitkommazahlen »float«, »double« und »long double«

Für reelle Zahlen wird der Basis-Fließkommadatentyp `float` verwendet. Dargestellt werden solche Zahlen durch eine ganze Zahl, die *Mantisse*, und den Exponent, der die Lage des Dezimalpunkts festlegt (Genaueres dazu gleich).

Mantisse

Mantisse ist ein lateinischer Begriff und die mathematische Bezeichnung für die Nachkommastellen.

Die Genauigkeit der Gleitkommadatentypen ergibt sich aus der Mantisse und ist abhängig vom Compiler. Wenn die Genauigkeit von `float` nicht mehr reicht, greift man auf `double` zurück. `double` weist mindestens die Genauigkeit von `float` auf. Wenn `double` auch nicht mehr ausreicht, verwendet man den Typ `long double` – der wiederum mindestens die Genauigkeit von `double` aufweist.

Beachten Sie bitte, dass C++ standardmäßig kein Dezimalkomma verwendet, wie Sie es gewohnt sind, sondern einen Dezimalpunkt. Dies hat allerdings eher herkunftsspezifische Gründe der Programmierung.

```
float a = 1,5;   // Komma ist falsch
float b = 5.1;   // richtig
```

Wenn einer der Werte vor oder hinter dem Dezimalpunkt 0 ist, beispielsweise `0.5` oder `1.0`, können Sie die Ziffer 0 auch weglassen. Der Compiler macht hieraus automatisch eine 0.

```
float c = .5;  // entspricht 0.5
float d = 5.;  // entspricht 5.0
```

Natürlich können Sie für eine Fließkommazahl auch einen Exponenten verwenden. Beispielsweise ist 1.2e34 die Abkürzung für $1.2 * 10^{34}$.

Hinweis **[«]**

Theoretisch ist es möglich, ein Komma statt eines Dezimalpunkts für Gleitpunktzahlen zu verwenden. Hierzu kann man ein `locale`-Objekt der gleichnamigen Header-Datei verwenden. Mit einem solchen Objekt können Sie die Sprachumgebung für das laufende Programm festlegen bzw. ändern.

Auch das Mischen von Integer und Gleitkommazahl ist erlaubt und möglich. Ist dabei der Divisor oder der Dividend eine Gleitkommazahl, wird automatisch eine Gleitkommadivision durchgeführt – also wird bei der Zuweisung einer Gleitkommazahl automatisch der Integer zu einer Gleitkommazahl konvertiert. Ein Beispiel dazu:

```
// float1.cpp
#include <iostream>
using namespace std;
int main(void) {
    float a = 3.0;
    float b = 2.0;
```

```
    int   c = 3;
    int   d = 2;
    cout << "3.0 / 2.0 = " << a/b << '\n';
    cout << "3   / 2   = " << c/d << '\n';
    cout << "3.0 / 2   = " << a/d << '\n';
    cout << "3   / 2.0 = " << c/b << '\n';
    return 0;
}
```

Das Programm bei der Ausführung:

```
3.0 / 2.0 = 1.5
3   / 2   = 1
3.0 / 2   = 1.5
3   / 2.0 = 1.5
```

Es ist natürlich auch erlaubt, einen Integer-Ausdruck einer Gleitkommavariablen zuzuweisen. Dabei wird der Integer-Ausdruck automatisch zu einer Gleitkommavariablen konvertiert. Das Gleiche gilt auch für die andere Richtung, nur dass Gleitkommazahlen, die Integern zugewiesen werden, ab dem Dezimalpunkt abgeschnitten werden.

Rechnen mit Gleitkommazahlen?

Vielleicht verwirrt Sie das Fragezeichen in dieser Überschrift. Computer sind doch die besten Rechner, und ich wage es hier, dies in Frage zu stellen. Der Umgang mit Integern ist für Computer kein allzu großes Problem, aber Gleitkommazahlen?

[»] **Hinweis**

Der Sinn dieses Abschnitts ist keineswegs, Ihnen nahezulegen, Gleitkommazahlen nicht in der Praxis zu verwenden, sondern ich will Sie nur auf einige Probleme hinweisen. Der Programmierer sind Sie selbst, und somit lastet die Verantwortung auch auf Ihren Schultern.

Zunächst möchte ich Ihnen das Gleitkommaformat beschreiben, das aus einem Vorzeichen, einem Dezimalbruch und einem Exponenten besteht.

$+\text{–}f.fff \times 10^{+\text{–}e}$

Zunächst finden Sie mit `+-` das Vorzeichen, gefolgt vom Dezimalbruch mit vier Stellen `f.fff` und am Ende den Exponent mit einer Stelle ($^{+\text{–}e}$). Die Zahlen werden gewöhnlich im E-Format ($+\text{–}f.fffE^{+\text{–}e}$) geschrieben. Zwar hat das IEEE das Gleitkommaformat standardisiert, aber leider halten sich nicht alle Computer daran. So wird die Zahl 1.0 wie im E-Format mit +1.000E+0 beschrieben oder –0.006321 mit –6.321E-3 und die 0.0 mit +0.000E+0.

So weit, so gut. Wenn Sie beispielsweise 2/6 + 2/6 rechnen, kommen Sie wohl auf das Ergebnis 4/6. Richtig für Sie, aber hier fängt das Dilemma Gleitkommazahlen und Rundungsfehler schon an. 2/6 ist im E-Format gleich +3.333E-1. Addieren Sie nun +3.333E-1 mit +3.333E-1, erhalten Sie als Ergebnis +6.666E-1 (bzw. 0.6666). Gut, aber leider falsch, denn 4/6 sind im E-Format +6.667E-1, aber nicht wie berechnet +6.666E-1. Mit derartigen Rundungsfehlern haben viele Computer ihre Probleme.

Daher hier nun der erste Ratschlag: Sollten Sie eine Software entwickeln, die mit Geldbeträgen arbeitet, und dabei auf keine anderen Bibliotheken bzw. BCD-Arithmetiken zurückgreifen wollen/können, dann sollten Sie die Beträge niemals im Gleitkommaformat verwenden (hierzulande also niemals in Euro und Cents, wie z. B. 1,99 €). Hier empfiehlt es sich, zur Berechnung die Geldbeträge in Cents als Integer-Zahl anzugeben, da bei immer intensiveren Berechnungen Rundungsfehler gemacht werden und somit eine falsche Berechnung garantiert ist.

Man kann nicht mal ganz genau sagen, wie genau eine solche gebrochene Zahl ist, weil dies von der Art der Berechnung abhängt. Beispielsweise führen Subtraktionen mehrerer ähnlicher Zahlen zu einem ungenaueren Ergebnis.

Um die Genauigkeit von `float` zu erhöhen, können Sie zum Beispiel `double` verwenden; damit erhalten Sie häufig die doppelte Genauigkeit. Auf vielen Rechnern hat `float` eine Genauigkeit von mindestens sechs Dezimalziffern und `double` häufig eine 15stellige Genauigkeit. Stellt man `double` noch das Schlüsselwort `long` voran, erhalten Sie gewöhnlich mindestens eine 19stellige Genauigkeit (auf HP-UX-Systemen – eine UNIX-Version von Hewlett Packard – gar eine 33stellige). Allerdings sind diese Angaben immer compilerabhängig, und die Rundungsfehler können trotzdem noch auftreten.

Um sich also mit den Problemen der Fließkommazahlen auseinanderzusetzen, müssen Sie sich mit Themen wie numerischer Analyse oder BCD-Arithmetik befassen. Allerdings sind dies Themen, die weit über dieses Buch hinausgehen würden.

Hinweis **[«]**

BCD steht für *Binary Coded Dezimals* und bedeutet, dass die Zahlen nicht binär, sondern als Zeichen gespeichert werden. Beispielsweise wird der Wert 56 nicht wie gewöhnlich als Bit-Folge 00111000 gespeichert, sondern als die Werte der Ziffern in dem jeweiligen Zeichensatz, in unserem Fall dem ASCII-Code-Zeichensatz. Und dabei hat das Zeichen »5« den Wert 53 und das Zeichen »6« den Wert 54. Somit ergibt sich folgende Bit-Stellung: 00110101 (53) 00110110 (54). Damit benötigt der Wert 53 allerdings 16 statt der möglichen acht Bits. Für die Zahl 12345 hingegen benötigen Sie schon 40 Bits. Es wird zwar erheblich mehr Speicherplatz verwendet, doch wenn Sie nur die Grundrechenarten für eine Ziffer implementieren, können Sie mit dieser Methode im Prinzip unendlich lange Zahlen bearbeiten. Es gibt keinen Genauigkeitsverlust.

1.4.7 Limits für Ganzzahl- und Gleitpunkt-Datentypen

In Abschnitt 1.4.5, »Die Datentypen ›int‹«, (beim Listing *int1.cpp*) haben Sie bereits erfahren, dass es für die einzelnen Datentypen eine Header-Datei gibt, in der Sie die minimalen bzw. maximalen Werte auf dem laufenden System abfragen können.

Die Limits für die Ganzzahltypen befinden sich in der Header-Datei <*climits*> (unter C bekannt als <*limits.h*>). Voraussetzung, um die einzelnen Werte in dieser Header-Datei abzufragen, ist natürlich, dass Sie diese Header-Datei mit einbinden:

Konstante	Erklärung
`CHAR_BIT`	Bit-Zahl für ein Byte
`SCHAR_MIN`	min. `signed char`
`SCHAR_MAX`	max. `signed char`
`UCHAR_MAX`	max. `unsigned char`
`CHAR_MIN`	min. `char`
`CHAR_MAX`	max. `char`
`MB_LEN_MAX`	max. Byte für ein Viel-Byte-Zeichen
`SHRT_MIN`	min. `short int`
`SHRT_MAX`	max. `short int`
`USHRT_MAX`	max. `unsigned short`
`INT_MIN`	min. `int`
`INT_MAX`	max. `int`
`UINT_MAX`	max. `unsigned int`
`LONG_MIN`	min. `long int`
`LONG_MAX`	max. `long int`
`ULONG_MAX`	max. `unsigned long int`

Tabelle 1.4 Limitkonstanten für ganzzahlige Datentypen in <climits>

Die Limitwerte für die Gleitkommazahlen finden Sie in der Header-Datei <*cfloat*> (unter C bekannt als <*float.h*>). Darin finden sich alle Konstanten und Eigenschaften, die für die Fließkommazahlen von Bedeutung sind. Natürlich müssen Sie hierbei auch die Header-Datei <*cfloat*> im Programm mit einbinden, wenn Sie einzelne Werte abfragen wollen:

Konstante	Bedeutung
`FLT_RADIX`	Basis für Exponentendarstellung
`FLT_MANT_DIG`	Anzahl Mantissenstellen (`float`)
`DBL_MANT_DIG`	Anzahl Mantissenstellen (`double`)

Tabelle 1.5 Limitkonstanten für Gleitpunkt-Datentypen in <cfloat>

Konstante	Bedeutung
LDBL_MANT_DIG	Anzahl Mantissenstellen (long double)
FLT_DIG	Genauigkeit in Dezimalziffern (float)
DBL_DIG	Genauigkeit in Dezimalziffern (double)
LDBL_DIG	Genauigkeit in Dezimalziffern (long double)
FLT_MIN_EXP	minimaler negativer FLT_RADIX-Exponent (float)
DBL_MIN_EXP	minimaler negativer FLT_RADIX-Exponent (double)
LDBL_MIN_EXP	minimaler negativer FLT_RADIX-Exponent (long double)
FLT_MIN_10_EXP	minimaler negativer Zehnerexponent (float)
DBL_MIN_10_EXP	minimaler negativer Zehnerexponent (double)
LDBL_MIN_10_EXP	minimaler negativer Zehnerexponent (long double)
FLT_MAX_EXP	maximaler FLT_RADIX-Exponent (float)
DBL_MAX_EXP	maximaler FLT_RADIX-Exponent (double)
LDBL_MAX_EXP	maximaler FLT_RADIX-Exponent (long double)
FLT_MAX_10_EXP	maximaler Zehnerexponent (float)
DBL_MAX_10_EXP	maximaler Zehnerexponent (double)
LDBL_MAX_10_EXP	maximaler Zehnerexponent (long double)
FLT_MAX	maximaler Gleitpunktwert (float)
DBL_MAX	maximaler Gleitpunktwert (double)
LDBL_MAX	maximaler Gleitpunktwert (long double)
FLT_EPSILON	kleinster float-Wert x, für den 1.0 + x ungleich 1.0 gilt
DBL_EPSILON	kleinster double-Wert x, für den 1.0 + x ungleich 1.0 gilt
LDBL_EPSILON	kleinster long double-Wert x, für den 1.0 + x ungleich 1.0 gilt
FLT_MIN	minimaler normalisierter Gleitpunktwert (float)
DBL_MIN	minimaler normalisierter Gleitpunktwert (double)
LDBL_MIN	minimaler normalisierter Gleitpunktwert (long double)

Tabelle 1.5 Limitkonstanten für Gleitpunkt-Datentypen in <cfloat> (Forts.)

1.4.8 Die Größen der Basistypen

Auch wenn hier häufig auf gewisse Größen der Basisdatentypen eingegangen wurde, so muss hinzugefügt werden, dass der C++-Standard hierfür keinerlei feste Größe vorschreibt. Die Größe von Basistypen wie bspw. short, int oder long ist immer implementierungsabhängig.

Hinweis [«]

Was der Standard hingegen vorschreibt, ist, dass ein char mindestens 8 Bit, ein short mindestens 16 Bit und ein long mindestens 32 Bit lang ist.

Fakt ist aber, dass die Größe der C++-Speicherobjekte immer als Vielfaches der Größe eines `char` verwendet wird. Und diese Größe ist per Definition gleich 1. Um die Größe des Typs zu ermitteln, wird in C++ der `sizeof`-Operator verwendet.

Es wird allerdings garantiert, dass der jeweilige »größere« Typ in der Reihenfolge `char`, `short`, `int`, `long`, `long long` den Wertebereich des »kleineren« Typs umfasst. Somit ist ein `char` niemals größer als ein `short`, `short` ist niemals größer als `int`, ein `int` niemals größer als ein `long` usw. Aus dieser Tatsache lassen sich die Größen der Typen folgendermaßen garantieren:

```
1==sizeof(char)<=sizeof(short)<=sizeof(int)<=sizeof(long)
sizeof(float) <= sizeof(double) <= sizeof(long double)
sizeof(bool) <= sizeof(long)
sizeof(char) <= sizeof(wchar_t) <= sizeof(long)
```

Weiterhin gilt, der Typ mit und ohne Vorzeichen (bspw. `signed int` und `unsigned int`) belegt exakt den gleichen Speicherplatz und verwendet auch die gleiche Anzahl von Bit. Dies garantiert somit Folgendes (`T` steht für den Typ):

```
sizeof(T) == sizeof(signed T) == sizeof(unsigned T)
```

Zugegeben, zunächst erscheint einem das Gerede um die Größe der Basistypen recht theoretisch und belanglos. Aber sobald Sie eine Software entwickeln müssen, bei der Code auf andere Systeme und Compiler portiert werden muss, kann das Missachten der Größe von Basistypen zu schwer auffindbaren Fehlern führen. Selbst wenn Sie planen, das Programm immer nur mit ein und demselben Compiler zu übersetzen, sollten Sie hier bedenken, dass auch eine künftige, neuere Version des Compilers auf einmal etwas anders macht als bisher.

1.4.9 void

In späteren Kapiteln werden Sie auch den fundamentalen Typ `void` kennenlernen. Es gibt allerdings keine `void`-Objekte. `void` kann nur für Funktionen (siehe Abschnitt 1.10, »Funktionen«) verwendet werden, die keinen Wert zurückgeben, oder auch als Zeiger auf Objekte mit einem unbekannten Typ (siehe Abschnitt 2.1.6, »void-Zeiger«). Folgendes ist also mit dem fundamentalen Typ `void` nicht möglich:

```
void obj;  // Falsch: void-Objekt nicht möglich
void &ref; // Falsch: Referenz auf void (siehe Abschnitt 2.2)
```

Richtig und möglich ist die folgende Verwendung von `void` (worauf allerdings in den entsprechenden Kapiteln noch eingegangen wird):

```
void function(); // Richtig: Funktion ohne Rückgabewert
void *vptr;      // Richtig: Zeiger auf unbekannten Typ
```

1.5 Konstanten

Benötigt man einen unveränderbaren Wert, können Sie eine Konstante verwenden. Der Sinn und Zweck einer solchen Konstante ist, dass der Wert zur Laufzeit des Programms nicht mehr verändert werden kann. Eine solche Konstante können Sie definieren, wenn Sie vor den eigentlichen Datentyp das Schlüsselwort `const` setzen:

```
const int var = 365;        // Integer-Konstante
const char dquote = '"';    // Zeichenkonstante
const float pi = 3.141592;  // Fließkommakonstante
```

Wenn nun aus Versehen versucht wird, den Wert dieser Konstante zu verändern, gibt der Compiler zur Übersetzungszeit einen Fehler aus, da es sich bei dieser Variablen um eine Read-only(nur lesbare)-Variable handelt. Somit können `const`-Werte auf gewöhnlichem Wege der direkten Zuweisung nicht mehr verändert werden.

In der Praxis werden Konstanten mit `const` recht häufig bei Variablen, Objekten, Zeigern, Parameter von Funktionen usw. verwendet, also immer dann, wenn ein Wert nicht mehr verändert werden darf.

Einen Vorteil kann man aus Konstanten auch ziehen, wenn der Compiler jede weitere Verwendung der Konstante bereits kennt. Dann muss dieser für das Objekt keinen Speicher mehr anlegen, weil die Konstante schon bei der Übersetzung ausgewertet werden kann. Ein Beispiel dazu:

```
const int day = 365;  // Zur Übersetzungszeit bekannt
const int month = 12; // Dito
const int current_day = curDay(); // Unbekannt
```

An den Deklarationen von `day` und `month` erkennt der Compiler die Werte, womit diese in konstanten Ausdrücken verwendet werden können. Dadurch, dass der Wert zur Übersetzungszeit bekannt ist, muss kein Speicher dafür angelegt werden. Da der Wert von `current_day` erst mit Hilfe der Funktion `curDay()` ermittelt werden muss, ist dieser Wert zur Übersetzung nicht bekannt, und somit muss hierfür auf jeden Fall Speicher angelegt werden.

Es muss allerdings auch erwähnt werden, dass nicht alle Compiler so intelligent sind und den Speicher sparen, wenn die Benutzung eines konstanten Objekts zur Übersetzungszeit bekannt ist. Dies ist somit immer abhängig von der Implementierung des Compilers.

[»] **Hinweis**

Wer sich mit der C-Programmierung beschäftigt hat, hat häufig auch Konstanten mit `#define` erzeugt. Leider werden diese Konstanten noch vor dem Compiler-Lauf vom Präprozessor durch den entsprechenden Text ersetzt. Das Problem dabei ist, dass der Ersetzungsvorgang nicht während der C++-Übersetzung stattfindet und somit die Syntax- und Typprüfung umgeht. Wo sich doch gerade C++ gegenüber C mit einer verbesserten Typprüfung rühmt, sollte man in C++ den Einsatz von `#define` für Konstanten ganz unterlassen und stattdessen `const` verwenden.

1.6 Standard Ein-/Ausgabe-Streams

In den vorangegangenen Beispielen wurde mehrfach `cout` zur Ausgabe auf dem Bildschirm verwendet, ohne dass darauf näher eingegangen wurde. Daher folgt jetzt eine Einführung in die Ein-/Ausgabe-Streams von C++.

Wer bereits Kenntnisse in C gesammelt hat, dürfte die Funktionen `scanf()` zur Eingabe und `printf()` zur Ausgabe auf dem Bildschirm kennen. Neben den Ein-/Ausgabefunktionen von C verfügt C++ über ein neues Ein-/Ausgabesystem, das (natürlich) auf der Basis von Klassen aufbaut. Aber keine Sorge, da bisher noch nicht auf die Klassen eingegangen wurde, benötigen Sie hierüber noch keine Kenntnisse, um diese Streams zu verstehen. Auf die Klassenhierarchie wird erst viel später bei der Erklärung der iostream-Bibliothek eingegangen.

[»] **Hinweis**

Wer sich für die Ein-/Ausgabefunktionen von C (und der Programmiersprache C überhaupt) und deren Einsatz interessiert, die ja auch in C++ verwendet werden können, dem sei mein Buch »C von A bis Z« empfohlen, das Sie auch online unter *http://www.pronix.de/* oder auf der Buch-CD finden.

Auch wenn man in C++ (fast) alles aus der C-Welt verwenden kann, so hat doch das neue Stream-Konzept (basierend auf Klassen) von C++ einen erheblichen Vorteil, nämlich Typsicherheit. Probleme, wie ein falsches Formatzeichen bei der Ausgabe von `printf()`, die zu mitunter gefährlichen Laufzeitfehlern geführt haben, gibt es mit den neuen Streams nicht mehr. Hierbei entscheidet nun der Compiler anhand des Argumenttyps, welche Ein- bzw. Ausgabefunktion aufzurufen ist.

1.6.1 Die neuen Streams – »cout«, »cin«, »cerr«, »clog«

Die neuen Streams sind in der Header-Datei <*iostream*> folgendermaßen deklariert (siehe Abbildung 1.2):

- `cout` – Standardausgabe (basiert auf dem C-Standard-Stream `stdout`)
- `cerr` – Standardfehlerausgabe (basiert auf dem C-Standard-Stream `stderr`)
- `clog` – Standardfehlerausgabe (wie `cerr`, nur gepuffert)
- `cin` – Standardeingabe (basiert auf dem C-Standard-Stream `stdin`)

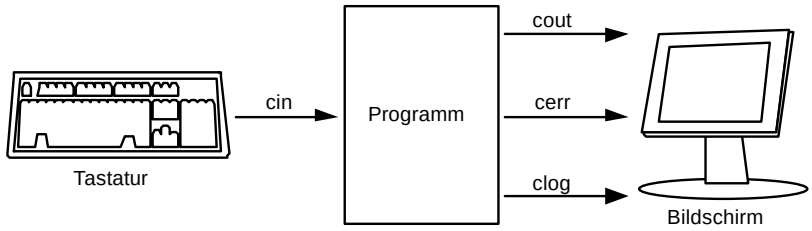

Abbildung 1.2 Ein- und Ausgabe-Streams in C++

Hinweis [«]

Auch wenn die neuen Streams auf den C-Standard-Streams basieren, sollte man möglichst nicht die »alten« C-Streams mit den neuen Streams mischen.

Hinweis [«]

Diese Streams basieren alle auf dem Datentyp `char`. Wollen Sie breite Zeichen (basierend auf `wchar_t`) ausgeben, stehen Ihnen die Streams `wout`, `werr`, `wlog` und `win` zur Verfügung.

Damit Sie die in der C++-Standardbibliothek definierten globalen Bezeichner zum Namensbereich `std` direkt verwenden können, müssen Sie die Direktive

```
using namespace std;
```

angeben (gewöhnlich am Anfang der Datei hinter der Angabe der Header-Dateien). Ohne Angabe des Namensbereichs `std` könnten Sie statt

```
cout << "Hallo Welt!\n";
```

nur mit dem Namensbereich `std` und dem Bereichsoperator `::` eine Ausgabe über den Stream `cout` machen:

```
std::cout << "Hallo Welt!\n";
```

Auf den Namensbereich und den Bereichsoperator wird allerdings erst in einem späteren Abschnitt eingegangen.

1.6.2 Ausgabe mit »cout«

Die Ausgabe mit `cout` wurde schon des Öfteren hier verwendet. Damit die Ausgabe realisiert wird, wurde der Operator `<<` (Shift-Operator) »überladen«, wodurch er für die Speicherobjekte dieser Klasse einen neuen Sinn erhält (zum Überladen auch später mehr).

```
cout << "Hallo Welt!\n";
```

Mit solch einer Anweisung schieben Sie den String »`Hallo Welt!`« in die Standardausgabe (`cout` alias `stdout`). Mit dem Operator `<<` zeigen Sie an, wohin der Text geschoben wird. Natürlich können Sie dahinter noch einen weiteren Text »schieben« lassen:

```
// Bspw. einen Text über zwei Zeilen
cout << "Hallo Welt!\n"
     << "Noch eine Zeile\n";
```

Sie können sich dies gerne wie bei einer Laufschrift vorstellen. Abgeschlossen wird die Ausgabe mit einem Semikolon.

Bei den Datentypen muss man nicht mehr C-typisch auf das richtige Formatzeichen achten, sondern hier ist `cout` clever genug, den Typ dieser Daten selbst zu erkennen, wie Sie dies bereits im Abschnitt zu den Basisdatentypen feststellen konnten. `cout` kann also alle elementaren Datentypen ausgeben.

1.6.3 Ausgabe mit »cerr«

`cerr` funktioniert im Grunde ähnlich wie `cout` und sendet seine Ausgabe auch auf den Bildschirm. Gewöhnlich verwendet man den Fehlerausgabe-Stream `cerr`, wenn man den Standardausgabe-Stream `cout` umgeleitet hat oder wenn man nur Fehler in eine Datei leiten will. Nur so kann man sicherstellen, dass nur noch die Fehlermeldungen auf dem Bildschirm ausgegeben werden.

Entscheidend ist auch zu wissen, dass `cerr` ungepuffert arbeitet, also die Ausgabe hier ungepuffert erfolgt. Aus Effizienzgründen ist die Ausgabe auf `cout` gewöhnlich gepuffert, das bedeutet, dass das System erst einmal x Zeichen zwischenspeichert und diese dann alle auf einmal ausgegeben werden.

[»] Hinweis

Man kann die Ausgabe mittels `flush` bzw. `cout.flush()` erzwingen.

1.6.4 Eingabe mit »cin«

Wollen Sie etwas von der Standardeingabe (Tastatur) einlesen, so erfolgt dies mit `cin` und dem Operator `>>`. Ein Beispiel hierfür:

```
// cin1.cpp
#include <iostream>
using namespace std;

int main(void) {
   float wert;
   cout << "Bitte eine Fließkommazahl : ";
   cin >> wert;
   cout << "Die Eingabe war " << wert << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
Bitte eine Fließkommazahl : 5.5
Die Eingabe war 5.5
```

Der Wert wird hierbei von `cin` in die Variable `wert` geschoben. Auch hierbei entfallen Fehler, die mit falschen Formatelementen bei `scanf()` schnell unterlaufen konnten. Am Typ der Variablen legt `cin` nämlich selbständig fest, was akzeptiert wird und wie die Konvertierung erfolgt. Beim Einlesen mit `cin` und dem Operator `>>` gelten außerdem folgende Regeln:

- Die Verarbeitung der Eingabe bricht an der Stelle ab, an der das erste Zeichen nicht mehr verarbeitet werden kann. Geben Sie im Listing *cin1.cpp* als Wert »5.5 Euro« an, so wird in der Variablen `wert` nur `5.5` gespeichert.
- Führende Leerraumzeichen wie Tabulator, Newline oder Whitespace werden überlesen. Befindet sich also vor dem eigentlichen Wert ein solches »Zeichen«, so wird der Wert dennoch richtig eingelesen.

In der Praxis sollten Sie sich aber nicht darauf verlassen, dass der Anwender schon das Richtige eingeben wird. Hierzu sollte man immer den Rückgabewert von `cin` und des Operators `>>` überprüfen. Bei einer richtigen Angabe wird `wahr` und bei einer falschen Angaben `falsch` zurückgegeben. Auf das Beispiel *cin1.cpp* bezogen, sieht eine solche Überprüfung wie folgt aus:

```
if( ! (cin >> wert) ) {
   cerr << "Fehler bei der Eingabe!\n";
}
else {
   cout << "Die Eingabe war " << wert << "\n";
}
```

Auf das `if-else`-Konstrukt wird später noch eingegangen, sofern Ihnen das noch nicht bekannt ist.

Einlesen von Zeichen und Zeichenketten

Wozu einen Extraabschnitt hierzu, werden Sie sich fragen. Weil es ein Problem sein kann, wenn beim Einlesen einzelner Zeichen ein führendes Leerzeichen eingegeben wurde und dies tatsächlich so gemeint war. Sie wissen ja, dass führende Leerzeichen vom Operator `>>` ignoriert werden. Wenn Sie also eine Eingabe zeichenweise einlesen wollen – also mit den führenden Leerraumzeichen –, dann benötigen Sie die `cin`-Methode `get()` (gleichwertig zur C-Anweisung `getchar()`).

```
// cin2.cpp
#include <iostream>
using namespace std;

int main(void) {
   int wert;
   cout << "Bitte ein Zeichen : ";
   wert = cin.get();
   cout << "Die Eingabe war   : " << (char)wert << '\n';
   return 0;
}
```

Es fällt auf, dass `cin` mit der Methode `get()` als Rückgabewert ein `int` zurückgibt. Dies muss so sein, weil `cin.get()` den Wert `EOF` für Dateiende oder einen Fehler zurückgibt. `EOF` kann auch mit der Tastenkombination Strg + Z (unter Windows/DOS) und Strg + D (unter Linux/UNIX) ausgelöst werden. Daher muss, sofern Sie nicht den ASCII-Wert des eingegebenen Zeichens haben wollen, der Wert mit `char` »gecastet« werden. Mehr zum Typcasting in einem gesonderten Abschnitt.

[»] Hinweis für Anfänger

Leider lässt es sich oft nicht vermeiden, das ein oder andere Thema hier schon mit einzubeziehen, das erst in einem späteren Abschnitt behandelt wird. Allerdings sollte Sie das nicht entmutigen, da hierauf noch zu gegebener Zeit eingegangen wird.

Hier haben Sie ein klassisches Grundgerüst (Listing *cin3.cpp*), wie es häufig bei einem Filterprogramm eingesetzt wird, das zeichenweise Text abarbeiten (oder gar verschlüsseln) soll.

```
// cin3.cpp
#include <iostream>
```

```
using namespace std;

int main(void) {
   int wert;
   // EOF kann mit STRG+Z bzw. STRG+D ausgelöst werden
   while( (wert = cin.get()) != EOF ) {
      cout << (char)wert;
   }
   return 0;
}
```

Wenn Sie dieses Beispiel ausführen, haben Sie im Grunde nichts anderes als einen Papagei, der alle Zeichen (beim Betätigen von ↵) ausgibt, die eingegeben wurden.

Ein ähnliches Problem entsteht übrigens auch, wenn Sie versuchen, einen String mit `cin` einzulesen. Befindet sich hier ein Zwischenraumzeichen, wie beispielsweise in »C++ von A bis Z«, so befinden sich im String nur die Zeichen »C++«. Das gleiche Verhalten kennen die C-Programmierer von der Funktion `scanf()`. Auch hier hat `cin` eine weitere Methode wie schon beim Einlesen einzelner Zeichen mit `get()`, und zwar die Methode `getline()`. Da auf die Strings noch gar nicht eingegangen wurde, nur schnell ein Codeausschnitt dazu, wie die Methode `getline()` in der Praxis verwendet wird:

```
char buch[20];
cin.getline( buch, 20 );
```

Beim Aufruf der Methode `getline()` müssen Sie auch die Größe mit angeben. Auf das Thema Strings wird noch in einem späteren Abschnitt – 7.1, »Die String-Bibliothek (string-Klasse)« – eingegangen.

1.7 Operatoren

Damit Sie anschließend nicht ins Straucheln geraten, sollen hier einige Begriffe zu den Operatoren im Voraus erläutert werden. Zunächst unterscheidet man Operatoren anhand der Anzahl ihrer Operanden:

- Unärer Operator – dieser Operator hat einen Operanden
- Binärer Operator – dieser Operator hat zwei Operanden
- Ternärer Operator – dieser Operator hat drei Operanden

In der Praxis werden Sie vorwiegend mit unären und binären Operatoren zu tun haben. Allerdings gibt es in C++ mit `?:` auch einen ternären Operator, aber dazu später mehr.

Neben der Anzahl von Operanden unterscheidet man auch die Position des Operators. Dabei verwendet man gewöhnlich drei verschiedene Begriffe:

- Präfix – der Operator steht vor dem Operanden
- Postfix – der Operator steht hinter dem Operanden
- Infix – der Operator steht zwischen den Operanden

Zum Schluss werden die Operatoren auch noch nach der Assoziativität differenziert. Als Assoziativität wird die Auswertungsreihenfolge bezeichnet, in der Operanden in einem Ausdruck ausgewertet werden. Dabei gibt es folgende Assoziativitäten der Operatoren:

- Linksassoziativität
- Rechtsassoziativität

Der Großteil der Operatoren in C++ ist linksassoziativ. Das bedeutet, dass zum Beispiel bei dem Ausdruck

```
var1 + var2 - var3;
```

zuerst `var1` mit `var2` addiert und `var3` anschließend von der Summe subtrahiert wird. Wären die Operatoren rechtsassoziativ, würde zuerst `var2` mit `var3` subtrahiert und danach erst mit `var1` addiert. Ist dies erwünscht, müssen Klammern gesetzt werden:

```
var1 + (var2 - var3);
```

1.7.1 Arithmetische Operatoren

Folgende arithmetische Operatoren sind in C++ vorhanden:

Operator	Bedeutung
+	Addiert zwei Werte (`var1+var2`).
–	Subtrahiert zwei Werte (`var1-var2`).
*	Multipliziert zwei Werte (`var1*var2`).
/	Dividiert zwei Werte (`var1/var2`).
%	Modulo (Rest einer Division) (`var1 %var2`)

Tabelle 1.6 Arithmetische Operatoren in C++

Es gelten für arithmetische Operatoren folgende (übliche mathematische) Regeln:

- Die klassische Punkt-vor-Strich-Regelung (`*` und `/` binden also stärker als `+` und `-`). In der Praxis heißt dies: 5 + 5 * 5 ergibt 30 und nicht, wie eventuell erwartet, 50. Wenn zuerst 5 + 5 berechnet werden soll, verwenden Sie Klammern. Diese binden dann stärker als die Rechenzeichen, also (5 + 5) * 5 = 50.
- Arithmetische Operatoren sind binäre Operatoren und haben somit immer zwei Operanden, also `<Operand><Operator><Operand>`.

Hinweis [«]

Grundsätzlich sollten Sie eine Division durch 0 vermeiden. Dies gilt sowohl für eine Division mit `/` als auch für den Modulo-Operator `%`. Der Programmabsturz ist Ihnen hierbei garantiert.

Hierzu ein recht einfaches Beispiel, das die arithmetischen Operatoren bei ihrer Anwendung in einem C++-Programm demonstriert.

```
// arith1.cpp
#include <iostream>
using namespace std;

int main(void) {
   int var1, var2, var3;

   cout << "Operand 1: ";
   cin >> var1;
   cout << "Operand 2: ";
   cin >> var2;

   // Berechnung direkt in cout
   cout << "Multiplikation: " << var1 << " * "
        << var2 << " = " << (var1*var2) << '\n';

   // Berechnung in var3 zwischenspeichern
   var3 = var1 + var2;
   cout << "Addition      : " << var1 << " + "
        << var2 << " = " << var3 << '\n';

   // Division durch 0 vermeiden
   (!var2) ?var2=1 :var2=var2;
   // Berechnung direkt in cout
   cout << "Division      : " << var3 << " / "
        << var2 << " = " << (var3/var2);
```

```
   // Den Rest der Division ermitteln
   cout << " (Rest : " << (var3%var2) << ")\n";

   // Neuen Wert von var1 zuweisen
   var1 = var3 - var2;
   cout << "Subtraktion   : " << var3 << " - "
        << var2 << " = " << var1 << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
Operand 1: 8
Operand 2: 3
Multiplikation: 8 * 3 = 24
Addition      : 8 + 3 = 11
Division      : 11 / 3 = 3 (Rest : 2)
Subtraktion   : 11 - 3 = 8
```

Eine Anmerkung noch zur Zeile

```
(!var2) ?var2=1 :var2=var2;
```

Hierbei (`?:`) handelt es sich um den einzigen ternären Operator in C++, der sich wie folgt beschreiben lässt:

```
(Ausdruck) ? Anweisung1 : Anweisung2
```

In Worten: Ist `Ausdruck` wahr, wird die `Anweisung1` ausgeführt, ansonsten die `Anweisung2`. Hierauf wird aber noch genauer eingegangen. Der Sinn dieser Zeile ist letztendlich zu vermeiden, dass Sie eine Division durch 0 ausführen. Wenn also der Ausdruck `var2` falsch ist (! = Negationsoperator), bekommt `var2` den Wert `1`, ansonsten bleibt alles beim Alten. Allerdings wird auf diesen Operator noch extra eingegangen.

Die arithmetischen Operatoren können Sie außerdem noch in einer erweiterten Darstellung verwenden:

Erweiterte Darstellung	Bedeutung
`+=`	`var1+=var2` ist gleichwertig zu `var1=var1+var2`.
`-=`	`var1-=var2` ist gleichwertig zu `var1=var1-var2`.
`*=`	`var1*=var2` ist gleichwertig zu `var1=var1*var2`.
`/=`	`var1/=var2` ist gleichwertig zu `var1=var1/var2`.
`%=`	`var1 %=var2` ist gleichwertig zu `var1=var1 %var2`.

Tabelle 1.7 Erweiterte Darstellung arithmetischer Operatoren

Die Schreibweise mit dem Operator und darauffolgendem = ist somit eigentlich nur eine kürzere Schreibweise und hat ansonsten gegenüber der üblichen Verwendung der arithmetischen Operatoren keine nennenswerten Vor- bzw. Nachteile.

1.7.2 Inkrement- und Dekrementoperator

Beim Inkrement- bzw. Dekrementoperator handelt es sich um unäre Operatoren mit nur einem Operanden. Die Operatoren machen nichts anderes, als den Wert einer Variablen um 1 zu erhöhen bzw. zu reduzieren. Beide Operatoren haben ihr Haupteinsatzgebiet in Schleifen. In C++ werden diese Operatoren wie folgt beschrieben:

Operator	Bedeutung
`++`	Inkrement (Variable um 1 erhöhen)
`--`	Dekrement (Variable um 1 verringern)

Tabelle 1.8 Inkrement- und Dekrementoperator

Zur Verwendung der Operatoren gibt es zwei verschiedene Schreibweisen. In der Praxis ist es von Bedeutung, welche davon man einsetzt:

Verwendung	Bezeichnung
`var++`	Postfix-Schreibweise
`++var`	Präfix-Schreibweise
`var--`	Postfix-Schreibweise
`--var`	Präfix-Schreibweise

Tabelle 1.9 Postfix- und Präfix-Schreibweisen

Folgende Unterschiede gibt es zwischen der Postfix- und der Präfix-Schreibweise:

- Die Postfix-Schreibweise erhöht bzw. verringert den Wert von `var`, gibt aber noch den alten Wert an den aktuellen Ausdruck weiter.
- Die Präfix-Schreibweise erhöht bzw. verringert den Wert von `var` und gibt diesen Wert sofort an den aktuellen Ausdruck weiter.

Hierzu ein einfaches Listing, das die beiden Operatoren im Einsatz demonstriert:

```
// incr1.cpp
#include <iostream>
using namespace std;
```

```
int main(void) {
   int var=1;
   // Inkrementoperator
   cout << "var=" << var << '\n';     // var=1
   var++;
   cout << "var=" << var << '\n';     // var=2
   cout << "var=" << var++ << '\n';   // var=2
   cout << "var=" << var << '\n';     // var=3
   cout << "var=" << ++var << "\n\n"; // var=4

   // analog dazu mit dem Dekrementoperator
   var--;
   cout << "var=" << var << '\n';     // var=3
   cout << "var=" << var-- << '\n';   // var=3
   cout << "var=" << var << '\n';     // var=2
   cout << "var=" << --var << '\n';   // var=1
   return 0;
}
```

Das Programm bei der Ausführung:

```
var=1
var=2
var=2
var=3
var=4

var=3
var=3
var=2
var=1
```

1.7.3 Bit-Operatoren

[»] **Hinweis**

Dieser Abschnitt setzt voraus, dass der Leser mit dem Dualsystem (bzw. Binärsystem) zur Speicherung vertraut ist. Mehr zum Dualsystem finden Sie im Anhang des Buches.

Wenn auf die binäre Darstellung von (Ganz-)Zahlen zugegriffen werden muss, kann dies mit Hilfe der Bit-Operatoren geschehen. Folgende Bit-Operatoren werden dazu angeboten:

Bit-Operator	Bedeutung
`&`, `&=`	bitweise AND-Verknüpfung
`|`, `|=`	bitweise OR-Verknüpfung (inkl.)
`^`, `^=`	bitweise XOR (exkl.)
`~`	bitweises Komplement
`>>`, `>>=`	Rechtsverschiebung
`<<`, `<<=`	Linksverschiebung

Tabelle 1.10 Übersicht über die bitweisen Operatoren

Sie finden hier neben der üblichen Schreibweise auch die erweiterte Zuweisungsschreibweise wieder. Natürlich sollte es klar sein, dass die Operatoren nur auf Ganzzahlen und nicht auf Gleitkommatypen anzuwenden sind.

Bitweises UND

Steht der `&`-Operator zwischen zwei Operanden, so handelt es sich um den bitweisen UND-Operator. Dieser wird gewöhnlich dazu verwendet, einzelne Bits gezielt zu löschen:

```
int var = 55;
var = var & 7;
```

Nach der Durchführung dieser Bit-Operation enthält `var` den Wert `7`. Um dies zu verstehen, sollten Sie die Regeln des bitweisen UND-Operators kennen:

var1	var2	var1&var2
0	0	0
0	1	0
1	0	0
1	1	1

Tabelle 1.11 Regeln einer bitweisen UND-Verknüpfung

Es werden alle Bits gelöscht, wenn nicht beide Operanden gesetzt sind. Sehen wir uns dazu die interne Bit-Darstellung der Berechnung von oben an (als 16-Bit-Darstellung), und verwenden wir hierzu die entsprechende Tabelle 1.10:

```
0000 0000 0011 0111   [ 55 ]
0000 0000 0000 0111   [  7 ]
---------------------------
0000 0000 0000 0111   [  7 ]
```

Bitweises ODER

Um gezielt einzelne Bits zu setzen, wird der bitweise ODER-Operator verwendet. Für den ODER-Operator gelten folgende Regeln in der Verknüpfungstabelle:

BitA	BitB	(BitA|BitB)
0	0	0
0	1	1
1	0	1
1	1	1

Tabelle 1.12 Regeln einer bitweisen ODER-Verknüpfung

In der Praxis sieht die Verwendung des bitweisen ODER-Operators wie folgt aus:

```
int var = 1;
var = var | 126;
```

Wendet man hierauf die Verknüpfungstabelle des bitweisen ODER-Operators an, kommt man folgendermaßen zum Ergebnis 127:

```
0000 0000 0000 0001   [ 1 ]
0000 0000 0111 1110   [126]
---------------------------
0000 0000 0111 1111   [127]
```

Bitweises XOR

Dieser exklusive ODER-Operator liefert nur dann 1 zurück, wenn beide Bits unterschiedlich sind. Dies wird gewöhnlich dazu verwendet, einzelne Bits umzuschalten – gesetzte Bits werden gelöscht und gelöschte gesetzt. Für die XOR-Verknüpfungen gelten folgende Regeln:

BitA	BitB	BitA^BitB
0	0	0
0	1	1
1	0	1
1	1	0

Tabelle 1.13 Regeln einer bitweisen XOR-Verknüpfung

Auch hierzu ein einfaches Beispiel:

```
int var = 20;
var = var ^ 55;
```

Wenden wir dazu die Regeln der Verknüpfungstabelle an, erhalten Sie als Ergebnis 35:

```
0000 0000 0001 0100    [ 20 ]
0000 0000 0011 0111    [ 55 ]
----------------------------
0000 0000 0010 0011    [ 35 ]
```

Bitweises Komplement

Der NOT-Operator (~) wirkt sich auf Zahlen so aus, dass er jedes einzelne Bit invertiert. Bei vorzeichenbehafteten Datentypen entspricht das einer Negation mit anschließender Subtraktion von 1:

```
int var = 20;
var =~ var;          // var = -21
```

Für den NOT-Operator gilt folgende Verknüpfungstabelle:

BitA	~BitA
0	1
1	0

Tabelle 1.14 Regeln einer bitweisen NOT-Verknüpfung

Links- bzw. Rechtsverschiebung

Bei einer Linksverschiebung mit << werden alle Bits einer Zahl um n Stellen nach links gerückt. Der auf der rechten Seite entstehende leere Teil wird mit 0 aufgefüllt. Aber Vorsicht! Wenn der Datentyp `signed` ist, dann ändert sich das Vorzeichen, wenn eine 1 in die Bit-Stelle des Vorzeichens gerückt wird. Wenn der linke Operand einen negativen Wert hatte, ist das Ergebnis compilerabhängig. Solche Linksverschiebungen werden gerne verwendet, um Zahlen zu potenzieren, denn rein mathematisch bedeutet eine solche Linksverschiebung eine Multiplikation mit 2. Bei Einrückung um zwei Stellen nach links wird mit 4 multipliziert, bei drei mit 8, bei vier mit 16 usw.

Solche Bit-Verschiebungen laufen erheblich schneller ab als normale arithmetische Berechnungen im Stil von 4*x. Allerdings sind die meisten Compiler heute selbst schlau genug, dies gegebenenfalls auch selbst zu tun.

Die Rechtsverschiebung mit dem >>-Operator ist das Gegenstück zur Linksverschiebung (<<). Damit können Sie statt einer Multiplikation mit 2 eine Division durch 2 bewirken. Ansonsten gilt dasselbe wie für die Linksverschiebung.

1.7.4 Weitere Operatoren

Natürlich gibt es noch eine Menge weiterer Operatoren, die zum gegebenen Zeitpunkt behandelt werden. Vorab ein kurzer Überblick über diese weiteren Operatoren:

Operator	Bedeutung
!	logisches Nicht
&&	logisches Und
||	logisches Oder (inkl.)
<	kleiner
>	größer
==	Gleichheit
!=	Ungleichheit
<=	kleiner-gleich
>=	größer-gleich

Tabelle 1.15 Weitere Operatoren im Überblick

1.8 Kommentare

Mit Kommentaren können Sie Ihren C++-Quelltext ein wenig beschreiben und lesbarer machen. Der Compiler ignoriert solche Kommentare und entfernt diese bei der Übersetzung vom Quelltext in die Maschinensprache. Sie müssen sich also keine Gedanken machen, dass ein vielkommentierter Quelltext den Umfang des ausführbaren Programms erhöht. Es stehen Ihnen in C++ zwei Möglichkeiten zur Verfügung, den Quelltext zu kommentieren:

- Kommentare in einer Zeile werden mit der Zeilenfolge `//` eingeleitet. Alles, was dahinter in dieser einen Zeile geschrieben wird, wird nun vom Compiler ignoriert.
- Kommentare über mehrere Zeilen werden zwischen den Zeichenfolgen `/*` und `*/` eingeschlossen. Das heißt, der Kommentar beginnt an der Stelle im Quellcode, an der die Zeichenfolge `/*` steht, und endet bei der Zeichenfolge `*/`. Diese Methode, Kommentare über mehrere Zeilen zu verwenden, wird gerne benutzt, um einen kompletten Block Quellcode »auszukommentieren«.

Was Sie kommentieren, bleibt Ihnen selbst überlassen. Allerdings sollte man nicht jede Zeile Code kommentieren oder Teile, die ohnehin klar sind. Wenn Sie versuchen, einen schwerverständlichen Code zu kommentieren, überlegen Sie vorher, ob es nicht vielleicht möglich ist, den Code zu vereinfachen.

1.9 Kontrollstrukturen

Bisher sind die Programme immer nur sequentiell abgelaufen – also immer Zeile für Zeile. In C++ haben Sie folgende drei Möglichkeiten, diesen sequentiellen Programmfluss zu verändern:

- Verzweigungen (oder auch Selektionen)
- Schleifen (oder auch Iterationen)
- Sprunganweisungen

1.9.1 Verzweigungen (Selektionen)

Mit einer Verzweigung können Sie den weiteren Ablauf des Programms von bestimmten Zuständen abhängig machen. Dabei wird im Programm eine Bedingung definiert, die entscheidet, an welcher Stelle das Programm fortgesetzt werden soll.

Die »if ... else«-Anweisung

Bei der `if`-Anweisung handelt es sich um eine einfache Verzweigung. Hier die Syntax:

```
if ( Bedingung ) {
   Anweisung(en);
}
```

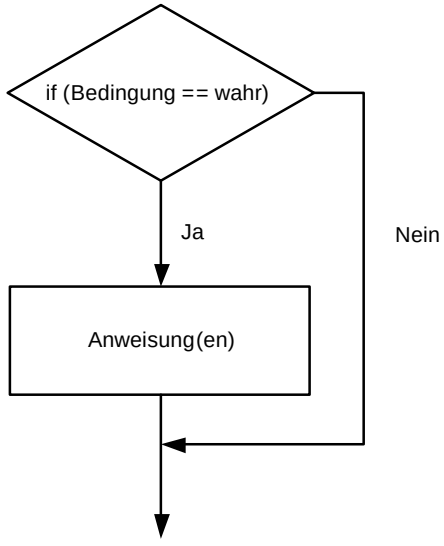

Abbildung 1.3 Programmablaufplan zur »if«-Anweisung

Beim Eintreten in die `if`-Anweisung wird zuerst die `Bedingung` überprüft. Wenn diese wahr (ungleich 0 oder auch `true`) ist, werden die `Anweisung(en)` im darauffolgenden Anweisungsblock ausgeführt, der durch geschweifte Klammern begrenzt wird, sofern er mehr als eine Zeile enthält. Wenn die Bedingung falsch ist (`false`), wird hinter dem Anweisungsblock mit der Programmausführung fortgefahren.

Hierzu ein einfaches Beispiel: Sie werden aufgefordert, zwei Ganzzahlen einzugeben, mit denen anschließend eine Division durchgeführt wird. Zunächst wird bei der Eingabe jeder Zahl überprüft, ob überhaupt eine gültige Zahl eingegeben wurde. Falls die Eingabe falsch war, gibt die Bedingung `false` (also unwahr/falsch) zurück. Anschließend wird noch überprüft, ob der Wert der Variablen `var2` gleich 0 ist – was bei einer Division niemals der Fall sein darf. Ist der Wert der Variablen `var2` gleich 0, ist die `if`-Anweisung wahr. In allen Fällen, in denen sich die Bedingung als wahr erweisen sollte, wird das Programm mit `exit` und dem Rückgabewert `1` beendet (natürlich mit entsprechender Ausgabe).

```
// if1.cpp
#include <iostream>
#include <cstdlib>
using namespace std;

int main(void) {
   int var1, var2;
   cout << "Bitte eine Zahl : ";
   if ( (cin >> var1) == false ) {
      cerr << "Fehler bei der Eingabe!\n";
      exit(1);
   }
   cout << "Bitte den Teiler eingeben : ";
   if ( (cin >> var2) == false ) {
      cerr << "Fehler bei der Eingabe!\n";
      exit(1);
   }
   if( var2 == 0 ) {
      cerr << "Teiler darf nicht 0 sein!\n";
      exit(1);
   }
   cout << "Ergebnis der Division " << var1 << " / "
        << var2 << " = " << (var1/var2) << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
Bitte eine Zahl : 10
Bitte den Teiler eingeben : a
Fehler bei der Eingabe!

Bitte eine Zahl : 10
Bitte den Teiler eingeben : 0
Teiler darf nicht 0 sein!

Bitte eine Zahl : 10
Bitte den Teiler eingeben : 2
Ergebnis der Division 10 / 2 = 5
```

Jeder `if`-Anweisung kann optional auch ein `else`-Zweig folgen. Dieser Teil wird immer dann ausgeführt, wenn die Auswertung der `if`-Bedingung falsch (`false`) war. Die Syntax dazu:

```
if ( Bedingung ) {
   Anweisung(en);
}
else {
   Anweisung(en);
}
```

Allerdings kann `else` niemals für sich allein stehen – `else` folgt immer einem vorausgehenden `if`.

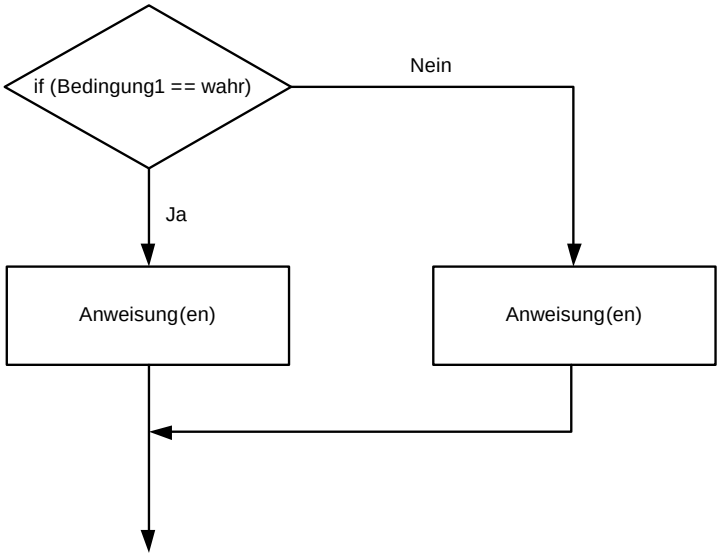

Abbildung 1.4 Programmablaufplan mit »else«

Ein einfaches Beispiel hierzu:

```
// if2.cpp
#include <iostream>
#include <cstdlib>
using namespace std;

int main(void) {
   int var;
   cout << "[0]   Ich will zu if\n";
   cout << "[1-9] Ich will zu else\n";
   cout << "\nIhre Wahl bitte : ";
   if( (cin >> var) == false ) {
      cerr << "Fehler bei der Eingabe!\n";
      exit(1);
   }
   if( var == 0 ) {
      cout << "Du hast if gewählt!\n";
   }
   else {
      cout << "Du hast nicht [0] gewählt daher else\n";
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
[0]   Ich will zu if
[1-9] Ich will zu else
Ihre Wahl bitte : 0
Du hast if gewählt!

[0]   Ich will zu if
[1-9] Ich will zu else
Ihre Wahl bitte : 4
Du hast nicht [0] gewählt daher else
```

Im Beispiel wurden die Blockanweisungen um den `if`- und `else`-Zweig verwendet:

```
   if( var == 0 ) {
      cout << "Du hast if gewählt!\n";
   }
   else {
      cout << "Du hast nicht [0] gewählt daher else\n";
   }
```

Besteht der jeweilige Zweig aus nur einer Anweisung wie in diesem Beispiel, so ist der Anweisungsblock nicht unbedingt erforderlich. Also könnte man diesen Zweig auch wie folgt angeben:

```
if( var == 0 )
   cout << "Du hast if gewählt!\n";
else
   cout << "Du hast nicht [0] gewählt daher else\n";
```

Rein »maschinell« sind beide identisch – also hat keine der beiden Methoden einen programmtechnischen Vorteil. Allerdings ist die Variante mit den Blockanweisungen leichter verständlich und lesbarer. Außerdem birgt die Methode ohne die Anweisungsblöcke die Gefahr, Anweisungen an der falschen Stelle zu schreiben und somit Fehler zu machen:

```
if( var == 0 )
   cout << "Du hast if gewählt!\n";
else
   cout << "Du hast nicht [0] gewählt daher else\n";
   cout << "Wo gehöre ich wohl hin?\n";
```

Wo gehört nun die Anweisung mit der Ausgabe `"Wo gehöre ich hin?"` hin? Rein programmiertechnisch gehört diese Anweisung nicht zum `else`-Zweig, aber war dies vom Programmierer auch so beabsichtigt? Anweisungsblöcke können in solch einem Fall Klarheit schaffen. Eine andere gut lesbare Möglichkeit wäre, die Anweisung direkt hinter die Bedingung zu schreiben:

```
if(var==0) cout << "Du hast if gewählt!\n";
else       cout << "Du hast nicht [0] gewählt daher else\n";
cout << "Wo gehöre ich wohl hin?\n";
```

Wenn Sie mehrere Bedingungen überprüfen wollen/müssen, können Sie auf `else if` zurückgreifen. Die Syntax ist folgende:

```
if ( Bedingung ) {
   Anweisung(en);
}
else if ( Bedingung ) {
   Anweisung(en);
}
// Optional
else {
   Anweisung(en);
}
```

Optional können Sie natürlich auch noch einen `else`-Zweig hinzufügen. Hierbei wird überprüft, ob die Bedingung der `if`-Anweisung wahr (`true`) ist. Wenn ja,

wird der `else if`-Zweig nicht mehr ausgeführt. Ansonsten, wenn die Bedingung der `if`-Anweisung falsch (`false`) ist, wird die Bedingung des `else if`-Zweigs ausgewertet. Trifft die Bedingung des `else if`-Zweigs zu (wahr/`true`), so werden die Anweisungen des entsprechenden Anweisungsblocks ausgeführt. Trifft keine der Bedingungen zu, wird das Programm hinter den Verzweigungen weitergeführt. Wenn eine `else`-Verzweigung vorhanden ist, werden die Anweisungen der `else`-Verzweigung ausgeführt (und anschließend wird mit der Programmausführung hinter den Verzweigungen fortgefahren). Natürlich können hierbei mehrere solcher `else if`-Verzweigungen folgen, wobei man in der Praxis oft auf die `switch`-Anweisung zurückgreift.

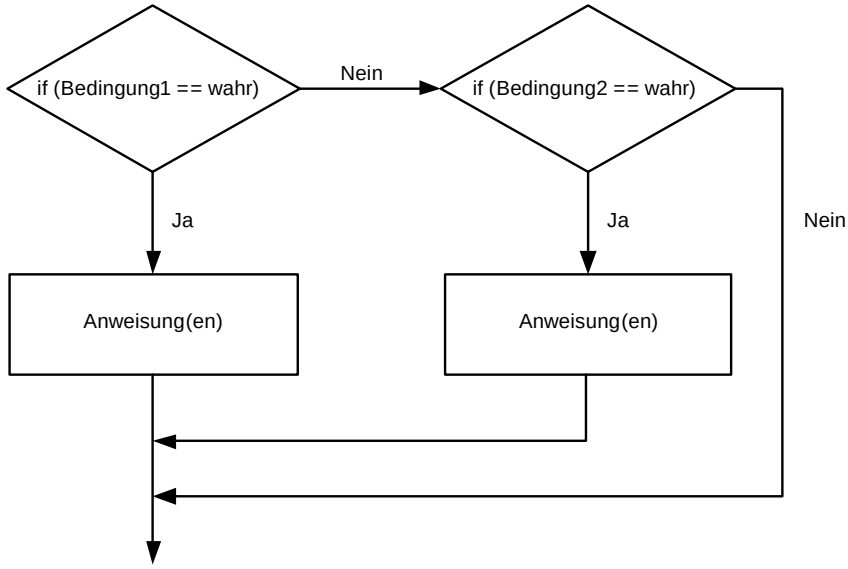

Abbildung 1.5 Programmablaufplan »if … else if«

Hierzu das Programm *if2.cpp* – durch `else if`-Zweige erweitert:

```
// if3.cpp
#include <iostream>
#include <cstdlib>
using namespace std;

int main(void) {
   int var;
   cout << "[0]   Ich will zu if\n";
   cout << "[1]   Ich will zu else if (1)\n";
   cout << "[2]   Ich will zu else if (2)\n";
   cout << "[3-9] Ich will zu else\n";
```

```
   cout << "\nIhre Wahl bitte : ";
   if( (cin >> var) == false ) {
      cerr << "Fehler bei der Eingabe!\n";
      exit(1);
   }
   if( var == 0 ) {
      cout << "Du hast if gewählt!\n";
   }
   else if ( var == 1 ) {
      cout << "Du hast else if (1) gewählt\n";
   }
   else if ( var == 2 ) {
      cout << "Du hast else if (2) gewählt\n";
   }
   else {
      cout << "Du hast nicht [0,1,2] gewählt daher else\n";
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
[0]   Ich will zu if
[1]   Ich will zu else if (1)
[2]   Ich will zu else if (2)
[3-9] Ich will zu else

Ihre Wahl bitte : 2
Du hast else if (2) gewählt
```

`if`-Anweisungen können natürlich auch verschachtelt werden – worauf man allerdings, wenn möglich, der Übersichtlichkeit halber verzichten sollte. Hier ein Ausschnitt aus dem Listing *if3.cpp*, nur »verschachtelter«:

```
   if( (cin >> var) ) {
      if( var == 0 ) {
         cout << "Du hast if gewählt!\n";
      }
      else if ( var == 1 ) {
         cout << "Du hast else if (1) gewählt\n";
      }
      else if ( var == 2 ) {
         cout << "Du hast else if (2) gewählt\n";
      }
      else {
         cout << "Du hast nicht [0,1,2] gewählt -> else\n";
```

```
      }
   }
   else {
      cerr << "Fehler bei der Eingabe!\n";
      exit(1);
   }
```

Immer noch lesbar, sagen Sie? Das Ganze kann man natürlich noch weitertreiben:

```
if( (cin >> var) )
   if( var != 0 )
      if ( var == 1 )
         cout << "Du hast else if (1) gewählt\n";
      else if ( var == 2 )
         cout << "Du hast else if (2) gewählt\n";
      else
         cout << "Du hast nicht[0,1,2]gewählt -> else\n";
   else
      cout << "Du hast if gewählt!\n";
else {
   cerr << "Fehler bei der Eingabe!\n";
   exit(1);
}
```

Hier wurde das Beispiel weiter verschachtelt, und zusätzlich wurden noch die nicht unbedingt erforderlichen Anweisungsblöcke entfernt. Mir persönlich graust es vor so einem Code, aber auch das ist eine Frage des persönlichen Stils.

Als Erweiterung von C wurde in C++ bei den Kontrollanweisungen `if`, `switch`, `for` und `while` ein eigener Gültigkeitsbereich eingeführt. Zwar wird auf dieses Thema noch gesondert eingegangen, aber es sollte hier schon mal erwähnt werden. Dieser Gültigkeitsbereich erstreckt sich vom Anfang der `if`-Anweisung bis zum Ende des Anweisungsblocks. Es kann also im Gegensatz zu C direkt in der `if`-Anweisung eine Variable deklariert werden:

```
if ( (int var = tage/stunden) > 9 ) {
   // Anweisungen -> hier ist var gültig
}
// Ab hier ist var nicht mehr gültig
```

Vergleichsoperatoren

In den Beispielen wurden schon häufig die Vergleichsoperatoren verwendet, um bestimmte Ausdrücke auszuwerten. Da in der Programmierung häufig reger

Gebrauch von diesen Operatoren gemacht wird, finden Sie hier eine Auflistung aller Vergleichsoperatoren in C++:

Vergleichsoperator	Bedeutung
`a < b`	wahr, wenn `a` kleiner als `b`
`a <= b`	wahr, wenn `a` kleiner oder gleich `b`
`a > b`	wahr, wenn `a` größer als `b`
`a >= b`	wahr, wenn `a` größer oder gleich `b`
`a == b`	wahr, wenn `a` gleich `b`
`a != b`	wahr, wenn `a` ungleich `b`

Tabelle 1.16 Übersicht über Vergleichsoperatoren (relationale Operatoren)

Hier werden oft Fehler beim Vergleichsoperator == gemacht, indem ein = vergessen wird.

```
if( var=100 ) {
   // Anweisung(en)
}
```

Rein syntaktisch gibt es hier nämlich gar keinen Fehler, weshalb sich Ihr Programm auch anstandslos ausführen lässt. Allerdings ist der Ausdruck `var=100` immer wahr, schließlich bekommt die Variable `var` den Wert `100` zugewiesen. Allerdings bin ich mir sicher, dass dies zu 99,9 % nicht so gewünscht wurde. Es ist recht leicht, sich davor zu schützen, indem man den Ausdruck einfach umdreht:

```
if( 100 == var ) {
   // Anweisung(en)
}
```

Würden Sie jetzt das =-Zeichen vergessen, läge ein Syntaxfehler vor, und der Compiler würde sich melden, da man einer Zahl keine Variable zuordnen kann.

Hinweis [«]

Wer sich fragt, warum man ausgerechnet das =-Zeichen zum Vergleich eines Ausdrucks verwendet, dem sei gesagt, dass die gute alte Bourne-Shell (sh) genau dies tut. Das Programm `test` alias `[` nutzt diese Syntax. Man kann diese Syntax somit theoretisch in jeder Shell nutzen, wenn `test` installiert ist (hier ist die Rede von linux-/unixartigen Systemen).

Bedingungsoperator »?:«

Den ternären Operator `?:` haben Sie ja schon einmal verwendet. Dieser stellt letztendlich nur eine Kurzform der `if ... else`-Anweisung dar. Die Syntax lautet:

```
(Ausdruck) ? (Anweisung 1) : (Anweisung 2)
```

Ist der `Ausdruck` wahr, dann wird die `Anweisung 1` ausgeführt – ansonsten wird zur `Anweisung 2` verzweigt. Will man zum Beispiel den größeren Wert von zwei Zahlen ermitteln, so lässt sich dies mit dem Operator `?:` wie folgt realisieren:

```
// tern.cpp
#include <iostream>
using namespace std;

int main() {
   int var1, var2, maxval;
   cout << "Bitte eine Zahl : ";
   cin >> var1;
   cout << "Noch eine Zahl  : ";
   cin >> var2;
   maxval = (var1 > var2) ?var1 : var2;
   cout << "Die größere Zahl lautet " << maxval << "\n";
   return 0;
}
```

Die äquivalente Schreibweise mit der `if ... else`-Anweisung sieht folgendermaßen aus:

```
if (a > b) {
   maxval = a;
}
else {
   maxval = b;
}
```

In solch einem Fall ist der ternäre Operator noch ganz vertretbar. Aber es lässt sich schon erkennen, dass dieser Operator nicht unbedingt dazu geeignet ist, einen klaren und gut leserlichen Code zu schreiben. Natürlich ist das auch wieder Ansichtssache und abhängig vom Codestil. Der folgende Ausschnitt soll verdeutlichen, was gemeint ist:

```
bignum = (a > b) ?((a > c) ?a :c) :((b > c) ?b :c);
```

Hier wurde der ternäre Operator verschachtelt, um den größten Wert von drei Zahlen zu ermitteln.

Logische Operatoren

Die logischen Operatoren in C++ sind `&&` (UND), `||` (ODER) und `!` (NICHT). Gewöhnlich werden logische Operatoren mit `bool`-Werten verwendet, aber praktisch kann man diese auch mit Zahlen und sonstigen Ausdrücken verknüpfen. In der Praxis lassen sich diese Operatoren auch überladen (später mehr dazu), wovon man aber Abstand nehmen sollte, um die Dinge nicht unnötig kompliziert zu gestalten.

Operator	Bedeutung
`&&`	logisches UND
`||`	logisches ODER
`!`	logisches NICHT

Tabelle 1.17 Die logischen Operatoren in C++

Ausdrücke, die durch den logischen ODER-Operator miteinander verknüpft sind, geben dann wahr (`true`) zurück, wenn mindestens einer der Ausdrücke wahr ist.

```
if ( (Bedingung1)  ||  (Bedingung2) ) {
   // mindestens ein Ausdruck ist wahr - true
}
else {
   // keiner der Ausdrücke ist wahr - false
}
```

Hierzu der Programmablaufplan des logischen ODER-Operators:

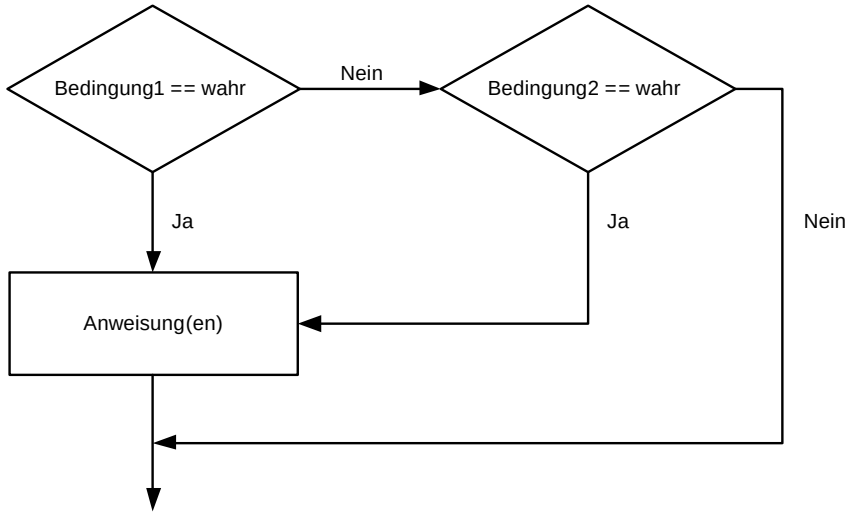

Abbildung 1.6 Programmablaufplan des logischen ODER-Operators

Anhand des Programmablaufplans (siehe Abbildung 1.7) lässt sich auch erkennen, dass der zweite Ausdruck gar nicht mehr ausgewertet wird, sobald der erste Ausdruck wahr ist. Daraus ergibt sich folgende logische Verknüpfungstabelle:

Bedingung1	Bedingung2	Bedingung1 || Bedingung2
true	true	true
true	false	true
false	true	true
false	false	false

Tabelle 1.18 Mögliche Ergebnisse einer logischen ODER-Verknüpfung

Hierzu ein einfaches Beispiel:

```
// logic_or.cpp
#include <iostream>
using namespace std;

int main(void) {
   int var1, var2;
   cout << "Bitte eine Zahl : ";
   cin >> var1;
   cout << "Noch eine Zahl  : ";
   cin >> var2;
   if( (var1 == 0) || (var2 == 0) ) {
      cerr << "Eine der Zahlen ist 0 !!!\n";
   }
   else {
      cout << "Keine der Zahlen hat den Wert 0\n";
   }
   return 0;
}
```

Sobald Sie im Beispiel eine Zahl mit dem Wert 0 belegen, gibt die logische ODER-Verknüpfung `true` zurück, und es erfolgt die entsprechende Ausgabe.

Anders hingegen sieht dies beim UND-Operator aus, der nur dann wahr (`true`) zurückliefert, wenn alle Ausdrücke wahr (`true`) sind. Die Syntax lautet:

```
if ( (Bedingung1)  &&  (Bedingung2) ) {
   // beide Ausdrücke sind wahr - true
}
else {
   // einer oder beide Ausdrücke sind falsch - false
}
```

Der Programmablaufplan des logischen UND-Operators ist in Abbildung 1.7 dargestellt.

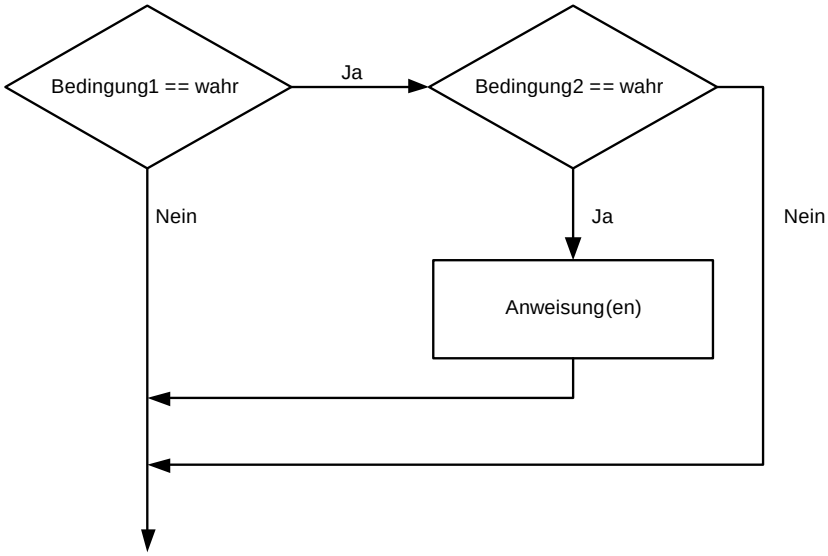

Abbildung 1.7 Programmablaufplan des logischen UND-Operators

Sofern der erste Ausdruck schon falsch (`false`) ist, wird der zweite Ausdruck gar nicht mehr ausgewertet, und es wird `false` zurückgegeben. Hierzu finden Sie die logische Verknüpfungstabelle des UND-Operators in Tabelle 1.19 wiedergegeben.

Bedingung1	Bedingung2	Bedingung1 && Bedingung2
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

Tabelle 1.19 Mögliche Ergebnisse einer logischen UND-Verknüpfung

Auch hierzu ein einfaches Beispiel:

```
// logic_and.cpp
#include <iostream>
using namespace std;

int main(void) {
   int var;
   cout << "Bitte eine Zahl von 1 bis 10: ";
   cin >> var;
```

```
   if( (var >= 1) && (var <= 10) ) {
      cout << "Danke !!!\n";
   }
   else {
      cerr << "Falsche Wert-Eingabe !!!\n";
   }
   return 0;
}
```

In diesem Beispiel werden Sie aufgefordert eine Zahl zwischen 1 und 10 einzugeben. Die logische UND-Verknüpfung überprüft, ob der Wertebereich der Zahl größer oder gleich 1 ist UND kleiner oder gleich 10. Trifft beides zu, wird `true` zurückgegeben, wenn einer der Ausdrücke falsch ist (`false`).

Selbstverständlich können Sie auch mehr als nur zwei Ausdrücke miteinander verknüpfen bzw. den `&&`- und den `||`-Operator kombinieren. Allerdings sollten Sie hierbei immer die Lesbarkeit des Quellcodes im Auge behalten.

Mit dem logischen NICHT-Operator (`!`) kann ein Ausdruck negiert werden. Man kann also aus `wahr falsch` machen und umgekehrt. Die folgende Tabelle demonstriert das:

Bedingung	!Bedingung
`true`	`false`
`false`	`true`

Tabelle 1.20 Ergebnisse des logischen NICHT-Operators

Häufig wird dieser Operator verwendet, um einige Dinge abzukürzen, z. B. um zu überprüfen, ob etwas 0 oder `false` ist. Vermutlich haben Sie schon oft Folgendes verwendet, um die richtige Eingabe des Typs zu überprüfen:

```
// gleichwertig zu if( (cin >> var) == 0 )
if ( (cin >> var) == false ) {
   cerr << "Falsche Eingabe - Keine Zahl\n";
   exit(1);
}
```

Dies lässt sich jetzt mit dem logischen NICHT-Operator wie folgt abkürzen:

```
if ( !(cin >> var) ) {
   cerr << "Falsche Eingabe - Keine Zahl\n";
   exit(1);
}
```

Natürlich ist der logische NICHT-Operator kein Operator, ohne den man nicht auskommen würde, wie die folgende Tabelle (1.21) zeigen soll:

Mit logischem NICHT	Ohne logisches NICHT

`if ( ! (ausdruck) )`	`if ( (ausdruck) == false )`
	`// oder`
	`if ( (ausdruck) == 0 )`

`if ( ! (var1 <= var2) )`	`if ( var1 > var2)`
`if(!(var <= 1) && !(var >= 11))`	`if((var >= 1) && (var <= 10))`

Tabelle 1.21 Vorgang mit und ohne den logischen NICHT-Operator

Es ist wohl immer eine Frage des Geschmacks und des Programmierstils.

Die »switch«-Anweisung

Die `switch`-Anweisung wird der `if`-Anweisung häufig vorgezogen, wenn einfache Abhängigkeiten von mehreren Zahlen oder Zeichen zur Verzweigung benötigt werden. Alles, was man mit der `switch`-Anweisung machen kann, könnte alternativ auch mit `if`-Anweisungen erledigt werden, wobei `switch` bei vielen Verzweigungen komfortabler und besser lesbar ist. Die Syntax zur `switch`-Anweisung lautet:

```
switch ( Bedingung ) {
   case Ausdruck_1 :
      // Anweisung(en)
      break;
   case Ausdruck_2 :
      // Anweisung(en)
      break;
   ...
   ...
   case Ausdruck_n :
      // Anweisung(en)
      break;
   default :
      // Anweisung(en)
}
```

Zunächst wird beim Eintreten der `switch`-Anweisung die Bedingung ausgewertet – dies muss ein integraler Ausdruck sein. Der nun folgende Anweisungsblock besteht aus einer Reihe von `case`-Marken. Jede dieser `case`-Marken stellt einen *Einsprungpunkt* dar. Hat `switch` die Bedingung ausgewertet, wird zu einem (falls vorhanden) passenden Einsprungpunkt verzweigt. Nun werden alle Anweisungen, die sich in diesem Einsprungpunkt befinden, abgearbeitet. Ist die letzte Anweisung des Einsprungpunkts (noch vor dem nächsten Einsprungpunkt) ein `break`, so wird aus dem `switch`-Anweisungsblock herausgesprungen (was

gewöhnlich der Fall ist). Sollten Sie kein `break` verwenden, so werden die dahinter folgenden Einsprungmarken ebenso ausgeführt (was durchaus gewollt sein kann).

Trifft keiner der Einsprungpunkte zu, wird entweder hinter der `switch`-Anweisung mit dem Programm fortgefahren oder, sofern eine Sprungmarke `default` vorhanden ist, zu dieser verzweigt. Natürlich darf es nur eine Sprungmarke mit dem Namen `default` geben. In der Praxis verwendet man in `switch`-Anweisungen immer einen `default`-Zweig für unvorhersehbare Fehler.

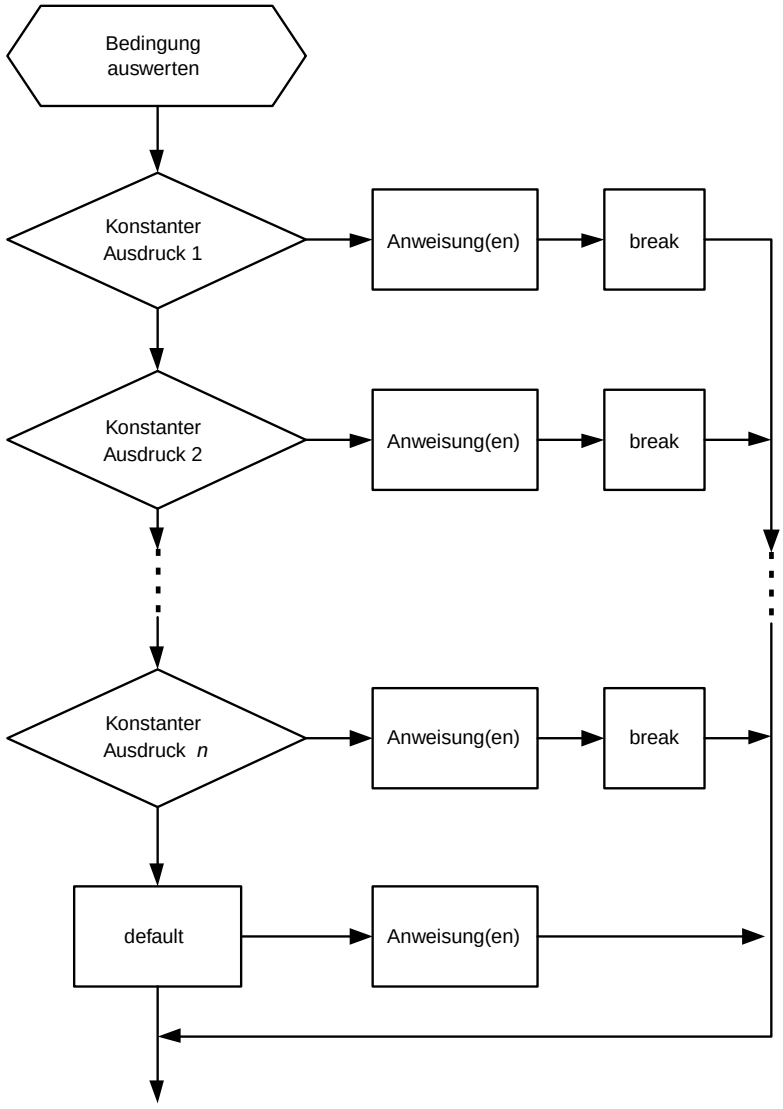

Abbildung 1.8 Programmablaufplan zur »switch«-Anweisung

Hierzu ein einfaches Beispiel:

```
// switch1.cpp
#include <iostream>
#include <cstdlib>
using namespace std;

int main(void) {
   int var;
   cout << "-1- Europa\n";
   cout << "-2- Asien\n";
   cout << "-3- Afrika\n";
   cout << "-4- Amerika\n";
   cout << "-5- Australien\n";
   cout << "Ihre Wahl bitte : ";
   if ( !(cin >> var) ) {
      cerr << "Falsche Eingabe - Keine Zahl\n";
      exit(1);
   }
   cout << "Ihre Wahl ist ";
   switch ( var ) {
      case 1 :
         cout << "Europa\n";
         break;
      case 2 :
         cout << "Asien\n";
         break;
      case 3 :
         cout << "Afrika\n";
         break;
      case 4 :
         cout << "Amerika\n";
         break;
      case 5 :
         cout << "Australien\n";
         break;
      default:
         cout << "\nFehler bei der Auswahl !\n";
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
-1- Europa
-2- Asien
```

```
-3- Afrika
-4- Amerika
-5- Australien
Ihre Wahl bitte : 3
Ihre Wahl ist Afrika

-1- Europa
-2- Asien
-3- Afrika
-4- Amerika
-5- Australien
Ihre Wahl bitte : 8
Ihre Wahl ist
Fehler bei der Auswahl !!!
```

Wie bereits erwähnt, ist es auch durchaus gängig, keinen »Aussprung« mit `break` zu machen (natürlich nur, sofern dies einen Sinn ergibt). Folgendes Beispiel soll das simulieren:

```
// switch2.cpp
#include <iostream>
#include <cstdlib>
using namespace std;

int main(void) {
   int var;
   cout << "-1- Produkt von A abholen\n";
   cout << "-2- Produkt auf Band legen\n";
   cout << "-3- Produkt vom Band abholen\n";
   cout << "-4- Produkt nach B bringen\n";
   cout << "Ihre Wahl bitte : ";
   if ( !(cin >> var) ) {
      cerr << "Falsche Eingabe - Keine Zahl\n";
      exit(1);
   }
   cout << "Ihre Wahl ist ";
   switch ( var ) {
      case 1 :
         cout << "Produkt von A abholen\n";
      case 2 :
         cout << "Produkt auf Band legen\n";
      case 3 :
         cout << "Produkt vom Band abholen\n";
      case 4 :
         cout << "Produkt nach B bringen\n";
```

```
         break;
      default:
         cout << "Fehler bei der Auswahl\n";
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
-1- Produkt von A abholen
-2- Produkt auf Band legen
-3- Produkt vom Band abholen
-4- Produkt nach B bringen
Ihre Wahl bitte : 1
Ihre Wahl ist Produkt von A abholen
Produkt auf Band legen
Produkt vom Band abholen
Produkt nach B bringen

-1- Produkt von A abholen
-2- Produkt auf Band legen
-3- Produkt vom Band abholen
-4- Produkt nach B bringen
Ihre Wahl bitte : 3
Ihre Wahl ist Produkt vom Band abholen
Produkt nach B bringen
```

In diesem Beispiel wird ein automatischer Vorgang simuliert. Hierbei wird ein Gegenstand zunächst von einem Ort »A« abgeholt, aufs Band gelegt, vom Band abgeholt und zu einem Ort »B« gebracht. Da es durchaus sein kann, dass die Maschine während eines Vorgangs angehalten wurde, kann nun mitten im Prozess darauf zugegriffen werden. Liegt ein Produkt bereits auf dem Band, und der Prozessvorgang wurde unterbrochen, dann kann dieser mit der Auswahl `3` von Hand zu Ende geführt werden.

Gerne und häufig wird die `switch`-Anweisung auch in Verbindung mit dem Auswerten (oder auch als Filter) einzelner Zeichen verwendet. Hier ein solches Beispiel, das alle Zeichen einliest, die Sie in der Standardeingabe eingeben, und die Zeichen `$` und `#` ausfiltert.

```
// switch3.cpp
#include <iostream>
using namespace std;

int main(void) {
   int ch;
```

```
   // EOF kann mit STRG+Z bzw. STRG+D ausgelöst werden
   while( (ch = cin.get()) != EOF ) {
      switch ( (char) ch) {
         case '$':
            // Hier auf das Dollarzeichen reagieren ...
            cout << "[Dollarzeichen]";
            break;
         case '#':
            // Hier auf das #-Zeichen reagieren ...
            cout << "[Hash-Zeichen]";
            break;
         // ... usw.
         default :
            cout << (char)ch;
      }
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Hallo Welt
Hallo Welt
Hier ein $-Zeichen
Hier ein [Dollarzeichen]-Zeichen
Und hier ein #-Zeichen
Und hier ein [Hash-Zeichen]-Zeichen
(Strg)+(Z) bzw. (Strg)+(D)
```

1.9.2 Schleifen (Iterationen)

Bei Schleifen (bzw. Iterationen) wird ein bestimmter Anweisungsblock so oft wiederholt, bis eine bestimmte Abbruchbedingung eintrifft (was auch nie der Fall sein kann).

Die »while«-Anweisung

Zunächst das Grundgerüst zur `while`-Anweisung:

```
while ( Bedingung ) {
   // Abarbeiten der Anweisungen
}
```

Die `while`-Schleife führt die Anweisungen, die im folgenden Anweisungsblock zusammengefasst sind, so lange aus, wie der Ausdruck der `while`-Anweisung wahr (`true`) ist. Ist der Ausdruck in der `while`-Anweisung hingegen falsch

(`false`), so wird der Anweisungsblock nicht (mehr) ausgeführt, und die Ausführung des Programms wird hinter dem Block fortgesetzt.

Üblicherweise durchläuft eine `while`-Schleife folgende Schritte:

- Initialisierung – die Schleifenvariable erhält ihren Anfangswert.
- Bedingung – der Ausdruck der Schleifenvariablen wird auf eine bestimmte Bedingung überprüft.
- Reinitialisierung – der Wert der Schleifenvariablen wird verändert.

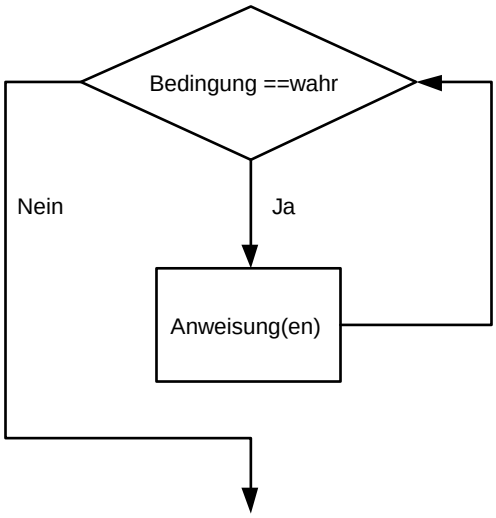

Abbildung 1.9 Programmablaufplan der »while«-Schleife

Rein programmtechnisch sieht dieser Vorgang folgendermaßen aus:

```
int var = 12;           // Initialisierung
while ( var < 12 ) {    // Ausdruck auf Bedingung überprüfen
   // Anweisungen
   var++;               // Reinitialisierung
}
```

Ein einfaches Beispiel zur `while`-Schleife:

```
// while1.cpp
#include <iostream>
using namespace std;

int main(void) {
   int var=1;
   while( var <= 6 ) {
```

```
      cout << var << ". Schleifendurchlauf\n";
      var++;
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
1. Schleifendurchlauf
2. Schleifendurchlauf
3. Schleifendurchlauf
4. Schleifendurchlauf
5. Schleifendurchlauf
6. Schleifendurchlauf
```

[»] **Hinweis**

Ein häufiger Fehler ist das Vergessen der Reinitialisierung für das Abbruchkriterium einer `while`-Schleife. Meistens kommt es dann ungewollt zu einer Endlosschleife, die nur noch »gewaltsam« unterbrochen werden kann.

Manchmal ist eine Endlosschleife durchaus erwünscht. Sie wird gerne eingesetzt, wenn etwas dauerhaft überwacht werden muss oder man auf ein bestimmtes Ereignis warten will. Ähnlich wird dies übrigens bei Bibliotheken mit einer grafischen Oberfläche gemacht, um auf Tastatur- oder Maus-Ereignisse zu warten, oder bei einer Serveranwendung in der Netzwerkprogrammierung – einfach überall, wo ein bestimmtes Server-/Client-Prinzip vorhanden ist. Im Grunde läuft eine solche Schleife immer wie folgt ab:

```
while ( true ) {
   bool ende = false;
   tue_etwas();
   if (eineBedingung()) {
      switch ( welcheBedingung ) {
         case Dies :
             // Anweisungen für Dies
             break;
         case Das :
             // Anweisungen für Das
             break;
         case Ende :
             // Anweisungen für Ende
             ende=true;
             break;
       }
   }
```

```
   // Endlosschleife abbrechen
   if ( ende == true ) {
      break;
   }
}
```

Die »do...while«-Anweisung

Die `do...while`-Anweisung wird ebenso wie die `while`-Anweisung ausgeführt, nur mit dem Unterschied, dass die Überprüfung der Bedingung erst nach dem Ende des Anweisungsblocks des Schleifenrumpfes stattfindet. Die Syntax lautet:

```
do {
   // Anweisungen
} while ( Bedingung );
```

Zunächst werden die Anweisungen im Schleifenrumpf ausgeführt. Anschließend wird der Ausdruck auf eine bestimmte Bedingung hin überprüft. Ist diese wahr (`true`), folgt ein erneuter Schleifendurchlauf. Trifft die Bedingung nicht mehr zu, wird die Programmausführung hinter dem Schleifenrumpf fortgeführt. Achten Sie auch darauf, dass Sie eine `do...while`-Schleife am Ende von `while` mit einem Semikolon abschließen.

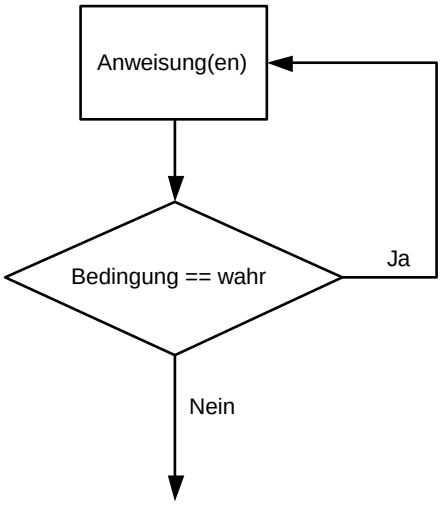

Abbildung 1.10 Programmablaufplan der »do...while«-Schleife

Da die `do...while`-Schleife den Schleifenrumpf mindestens einmal durchläuft, sollte man sie auch nur dann verwenden, wenn dies mindestens einmal der Fall sein soll.

Die `do...while`-Schleife lässt sich zum Beispiel hervorragend für einfache Benutzermenüs in der Konsole einsetzen, da eine solche mindestens einmal durchlaufen wird.

```
// do_while1.cpp
#include <iostream>
#include <cstdlib>
using namespace std;

int main(void) {
   int var;
   do {
      cout << "-1- Europa\n";
      cout << "-2- Asien\n";
      cout << "-3- Afrika\n";
      cout << "-4- Amerika\n";
      cout << "-5- Australien\n";
      cout << "-6- Ende\n";
      cout << "Ihre Wahl bitte : ";
      if ( !(cin >> var) ) {
         cerr << "Falsche Eingabe - Keine Zahl\n";
         exit(1);
      }
      cout << "Ihre Wahl ist ";
      switch ( var ) {
         case 1 :
            cout << "Europa\n";
            break;
         case 2 :
            cout << "Asien\n";
            break;
         case 3 :
            cout << "Afrika\n";
            break;
         case 4 :
             cout << "Amerika\n";
             break;
         case 5 :
            cout << "Australien\n";
            break;
         case 6 :
            cout << "\nProgramm wird beendet ...!\n";
            break;
         default:
            cout << "\nFehler bei der Auswahl !!!\n";
      }
```

```
   }  while( var != 6 );
   return 0;
}
```

Das Programm bei der Ausführung:

```
-1- Europa
-2- Asien
-3- Afrika
-4- Amerika
-5- Australien
-6- Ende
Ihre Wahl bitte : 2
Ihre Wahl ist Asien
-1- Europa
-2- Asien
-3- Afrika
-4- Amerika
-5- Australien
-6- Ende
Ihre Wahl bitte : 6
Ihre Wahl ist
Programm wird beendet ...!
```

Die »for«-Anweisung

Die `for`-Anweisung ist im Prinzip eine verkürzte `while`-Anweisung – mit dem Unterschied, dass bei der `for`-Anweisung die Initialisierung, die Auswertung der Bedingung und die Reinitialisierung schon in der `for`-Anweisung erfolgen können. Die Syntax lautet:

```
for( Initialisierung(en); Bedingung; Reinitialisierung(en) {
   // Anweisungen
}
```

Zunächst können bei der `for`-Schleife eine oder mehrere Variablen initialisiert werden. Dieser Vorgang geschieht allerdings einmalig – egal, wie oft der Schleifenrumpf anschließend ausgeführt wird. Nach der Initialisierung der Schleifenvariablen wird typischerweise der Ausdruck der Bedingung darauf überprüft, ob wahr (`true`) zurückgeliefert wird. Wird `true` zurückgegeben, werden die Anweisungen im Anweisungsblock der `for`-Schleife ausgeführt. Wenn alle Anweisungen ausgeführt wurden, werden die Reinitialisierungen der Schleifenvariablen ausgeführt. Anschließend wird erneut die Bedingung überprüft, und der Schleifendurchlauf beginnt gegebenenfalls von vorn. Der Rumpf der `for`-Schleife wird so oft durchlaufen, bis die Bedingung falsch (`false`) zurückgibt.

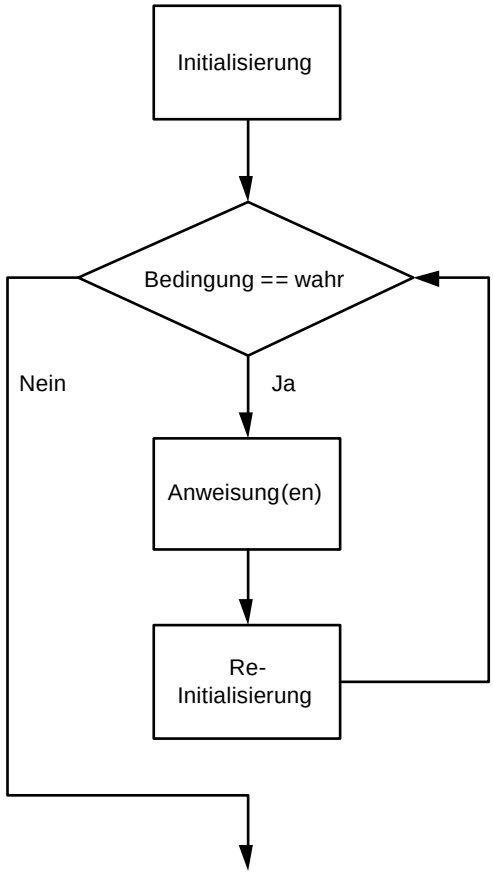

Abbildung 1.11 Programmablaufplan der »for«-Schleife

Hierzu ein einfaches Beispiel:

```
// for1.cpp
#include <iostream>
using namespace std;

int main(void) {
   int i;
   for ( i=1; i <= 6; i++ ) {
      cout << i << ". Schleifendurchlauf\n";
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
1. Schleifendurchlauf
2. Schleifendurchlauf
3. Schleifendurchlauf
4. Schleifendurchlauf
5. Schleifendurchlauf
6. Schleifendurchlauf
```

Das Programm macht also nichts anderes als schon das Beispiel *while1.cpp*. Es ist übrigens (in C++) durchaus üblich, die Schleifenvariable lokal zu deklarieren:

```
for ( int i=1; i <= 6; i++ ) {
   cout << i << ". Schleifendurchlauf\n";
}
// Hier ist i nicht mehr gültig
```

Die Variable ist somit nur innerhalb des Schleifenrumpfs gültig und verschwendet keinen unnötigen Speicherplatz.

Es ist außerdem auch möglich, mehrere Variablen in der `for`-Schleife zu initialisieren und gegebenenfalls auch zu reinitialisieren, um somit gleichzeitig zwei Indizien zu behandeln. Dabei werden die einzelnen Variablen mit einem einfachen Komma voneinander getrennt:

```
// for2.cpp
#include <iostream>
using namespace std;

int main(void) {
   for ( int n1=1, n2=2; n1 <= 10; n1++, n2*=2 ) {
      cout << (n1*n2) << '\n';
   }
   return 0;
}
```

Natürlich müssen Sie keineswegs alle Angaben der `for`-Schleife ausfüllen. Lassen Sie die Auswertung der Bedingung weg, so haben Sie praktisch eine Endlosschleife, die immer wahr ist.

```
// Endlosschleife
for (;;) {
   // Anweisungen
}
```

Soll die Schleife nur eine Bedingung auswerten, so lässt sich das auch mit `for` ohne Probleme bewerkstelligen:

```
for (; Bedingung == wahr; ) {
   // Anweisungen
}
```

Allerdings greift man in solchen Fällen doch eher zur `while`-Anweisung.

1.9.3 Sprunganweisungen

Bei Sprunganweisungen wird die Programmausführung mit Hilfe von Sprungmarken an einer anderen Position fortgesetzt. Obwohl nach wie vor möglich, werden Sprünge in einem Programm mittlerweile als schlechter Stil angesehen, und sie sind auch nicht notwendig. Die Rede ist von direkten Sprüngen. Die direkten Sprünge mit der `goto`-Anweisung werden in diesem Buch nicht mehr behandelt, da es wohl kein Problem dieser Welt gibt, das nicht ohne `goto` zu lösen wäre. Allein schon die Möglichkeit, Deklarationen mit Initialisierungen zu überspringen, ist für mich ein Argument, hierauf zu verzichten.

Mit Schlüsselwörtern wie `return`, `break`, `continue` und `exit` können jedoch auch kontrollierte Sprünge ausgeführt werden. Natürlich muss man dazu anmerken, dass `return` und `exit` eigentlich keine schleifentypischen Anweisungen sind, daher werden diese auch erst später behandelt.

Die »break«-Anweisung

Die `break`-Anweisung haben Sie bisher schon öfter angewandt. Mit `break` führen Sie einen Sprung aus der direkt umfassenden Schleife (im Anweisungsblock) oder aus der `switch`-Anweisung heraus aus. Oder einfacher gesagt: Mit `break` wird generell ein Anweisungsblock beendet – unabhängig von Schleife oder Verzweigung.

Das folgende Beispiel liest so lange Zeichen von der Tastatur ein und gibt diese auf die Standardausgabe aus, bis ein Punkt eingegeben wird. Wurde ein Punkt eingegeben, wird die Schleife mit `break` beendet, und die Ausführung des Programms wird hinter dem Schleifenrumpf fortgeführt.

```
// break1.cpp
#include <iostream>
using namespace std;

int main(void) {
   int ch;
   cout << " > ";
   while( (ch = cin.get()) != EOF ) {
      if( (char)ch == '.' ) {
         break;
```

```
      }
      if( (char)ch == '\n' ) {
         cout << "\n > ";
      }
      else {
         cout << (char) ch;
      }
   }
   cout << "Programmende erreicht!\n";
   return 0;
}
```

Das Programm bei der Ausführung:

```
> Hallo Welt
Hallo Welt
 > Jetzt folgt dann das Ende
Jetzt folgt dann das Ende
 > Hier der Punkt.
Hier der Punkt
Programmende erreicht!
```

Beachten Sie bitte, dass beim Setzen von `break` in einer verschachtelten Schleife immer nur die innerste Schleife abgebrochen wird.

Die »continue«-Anweisung

Im Gegensatz zu `break` beendet die `continue`-Anweisung nur die aktuelle Schleifenausführung. Das heißt, es werden alle noch hinter `continue` folgenden Anweisungen ausgelassen, und es wird mit der Programmausführung zum nächsten Schleifendurchlauf gesprungen.

Bauen Sie die `continue`-Anweisung anstelle der `break`-Anweisung im Beispiel *break1.cpp* ein, wird zurück zum Schleifenanfang gesprungen, anstatt den Schleifendurchlauf zu beenden; der Punkt wird also einfach ignoriert und nicht ausgegeben.

```
// continue1.cpp
#include <iostream>
using namespace std;

int main(void) {
   int ch;
   cout << " > ";
   while( (ch = cin.get()) != EOF ) {
      if( (char)ch == '.' ) {
```

```
         continue;
      }
      if( (char)ch == '\n' ) {
         cout << "\n > ";
      }
      else {
         cout << (char) ch;
      }
   }
   cout << "Programmende erreicht!\n";
   return 0;
}
```

Das Programm bei der Ausführung:

```
> Hallo Welt
Hallo Welt
 > Wie gehts.
Wie gehts
 > Ein toller Tag.
Ein toller Tag
 > .....

 > (Strg)+(D) oder (Strg)+(Z)
Programmende erreicht!
```

Gerade in Verbindung mit Schleifenvariablen sollten Sie bei der Verwendung von `continue` Vorsicht walten lassen. Das folgende Beispiel wird sich nicht mehr ohne äußere Gewalt beenden lassen, weil die Schleifenvariable nicht mehr inkrementiert wird:

```
// continue2.cpp
#include <iostream>
using namespace std;

int main(void) {
   int var=0;
   while( var < 10) {
      if( var % 2 ) {
         continue;
      }
      var++;
   }
   return 0;
}
```

Sobald der Wert von `var` mit dem Modulo-Operator (`%`) durch zwei geteilt wird und die Bedingung der `if`-Anweisung `true` ist, wird sich die Schleife nicht mehr von selbst beenden können, weil die Zeile `var++` niemals erreicht wird und `var % 2` immer wahr ist. Wenn die Bedingung `true` zurückgibt, bedeutet dies, dass die Zahl ungerade ist, da die Division einen Rest enthält.

1.10 Funktionen

Funktionen sind praktisch kleine Unterprogramme (oder auch benutzerdefinierte Operationen), mit denen Sie Daten verarbeiten oder Teilprobleme lösen können. In den Beispielen zuvor haben Sie bereits immer eine Funktion verwendet, nämlich die `main`-Funktion. Egal, wie viele Funktionen Sie schreiben, es wird immer eine `main`-Funktion benötigt. Die `main`-Funktion ist außerdem immer die erste Funktion, die beim Programmstart aufgerufen wird. Den ersten Funktionsaufruf werden Sie daher immer von der `main`-Funktion aus machen, alle weiteren Aufrufe können dann auch von den aufgerufenen Funktionen aus gestartet werden.

Wenn Sie zum Beispiel eine komplizierte Berechnung erstellen müssen, werden Sie dies wohl kaum in der `main`-Funktion machen. Dazu schreiben Sie eine gesonderte Funktion. Aus der `main`-Funktion rufen Sie dann die Funktion (gegebenenfalls mit den Argumenten) auf. Sie berechnet dann die Aufgabe oder zumindest einen Teil davon und gibt den Wert entweder an den Aufrufer zurück oder ruft gegebenenfalls eine weitere Funktion auf, um beispielsweise das Ergebnis der Berechnung in einer Datei zu speichern usw.

In der Praxis sollten Sie die Aufgaben möglichst auf viele kleine Funktionen aufteilen – eine Funktion soll ihre Aufgabe erfüllen und fertig. Wenn eine weitere Funktionalität benötigt wird, kann eine weitere Funktion geschrieben werden. So lässt sich der Code besser lesen, verbessern und gegebenenfalls auch wiederverwenden.

Leider gibt es in C++ keinen standardisierten Weg, Funktionen parallel auszuführen (Stichwort *Multithreading*). Hierzu müssen Sie auf externe Bibliotheken zurückgreifen. Die Funktionen in C++ laufen in der Regel sequentiell ab, also nacheinander.

1.10.1 Deklaration und Definition

Bevor Sie eine Funktion verwenden bzw. aufrufen können, müssen Sie diese deklarieren und dann erst definieren. Mit der Deklaration teilen Sie dem Compiler den Namen der Funktion, den Typ des zurückzugebenden Funktionswerts

und die Parameter der Funktionen mit. Die Deklaration einer Funktion wird als *Prototyp* bezeichnet.

Die Syntax einer solchen Deklaration sieht wie folgt aus:

```
[Spezifizierer] Rückgabetyp Funktionsname(Parameter);
```

Somit ist eine solche Deklaration wie folgt gegliedert:

- Rückgabetyp – hier legen Sie den Datentyp des Rückgabewerts fest. Dabei dürfen alle bisher kennengelernten Datentypen verwendet werden. Eine Funktion ohne Rückgabewert wird als `void` deklariert.
- Funktionsname – dies ist ein eindeutiger Funktionsname, mit dem Sie die Funktion von einer anderen Stelle aus im Programmcode aufrufen können. Für den Funktionsnamen selbst gelten dieselben Regeln wie für Variablen. Außerdem sollten Sie keine Funktionsnamen der Laufzeitbibliothek verwenden.
- Parameter – die Parameter einer Funktion sind optional. Sie werden durch den Datentyp und den Namen spezifiziert und durch ein Komma getrennt. Wird kein Parameter verwendet, können Sie zwischen die Klammern entweder `void` oder gar nichts schreiben.
- Spezifizierer – außerdem lassen sich bei Funktionen auch sogenannte Speicherklassen-Spezifizierer verwenden. Mehr dazu finden Sie im entsprechenden Abschnitt.

Hier eine solche Funktionsdeklaration:

```
int checkPoint ( int xPos, int yPos );
```

Beim Prototyp der Funktion müssen Sie bei den Funktionsparametern allerdings den Namen nicht zwangsläufig angeben, sondern nur die Datentypen:

```
int checkPoint ( int, int );
```

Allerdings lässt sich an der ersten Deklaration wesentlich schneller ablesen, was die einzelnen Parameter der Funktion bedeuten, wenn man trotzdem einen Namen verwendet. Der folgende Codeausschnitt soll eine solche Funktionsdeklaration näher demonstrieren:

```
// func1.cpp
#include <iostream>
using namespace std;

// Funktions-Prototyp (Deklaration)
int checkPoint ( int xPos, int yPos );
```

```
int main(void) {
   // Anweisungen
   return 0;
}
```

Hinweis [«]

Jede Funktion muss vor der Anwendung deklariert werden. Im Gegensatz zu den Variablen stellt eine Funktionsdeklaration noch keine Definition dar – hierzu fehlt noch der eigentliche Code der Funktion.

Mit der jetzt folgenden Definition teilen Sie dem Compiler die eigentliche Arbeitsweise der Funktion mit. Eine Funktionsdefinition enthält neben den Daten vom Prototyp auch den Funktionsrumpf (bzw. den Anweisungsblock), in dem die Anweisungen (die eigentliche Arbeit) der Funktion geschrieben werden. Außerdem benötigt eine Funktion kein abschließendes Semikolon mehr. Die Syntax der Funktionsdefinition sieht demnach wie folgt aus:

```
[Spezifizierer] Rückgabetyp Funktionsname(Parameter) {
   // Anweisungen
}
```

Somit sieht eine komplette Deklaration und Definition einer Funktion folgendermaßen aus:

```
// func1.cpp
#include <iostream>
using namespace std;
// Funktions-Prototyp (Deklaration)
int checkPoint ( int xPos, int yPos );

int main(void) {
   // Anweisungen
   return 0;
}

// Funktionsdefinition
int checkPoint ( int xPos, int yPos ) {
   // Anweisungen
}
```

Die Position der Funktionsdefinition muss allerdings nicht zwangsläufig hinter der `main`-Funktion stehen, sondern darf natürlich auch davor platziert werden:

```
// func1.cpp
#include <iostream>
using namespace std;
```

```
// Funktions-Prototyp (Deklaration)
int checkPoint ( int xPos, int yPos );

// Funktionsdefinition
int checkPoint ( int xPos, int yPos ) {
   // Anweisungen
}

int main(void) {
   // Anweisungen
   return 0;
}
```

Dies ist wohl wieder eher eine Frage des Programmierstils. Mir persönlich fällt es mittlerweile leichter, die Funktionsdefinition hinter der `main`-Funktion zu setzen, weil ich hierbei, angefangen bei `main()`, den Programmablauf besser verfolgen kann, sofern die Funktionen und `main()` in derselben Datei stehen, was in der Praxis eher selten der Fall ist. Aber dazu in einem späteren Abschnitt mehr.

1.10.2 Funktionsaufruf und Parameterübergabe

Wenn Sie den Code für die Funktion definiert haben, können Sie diese aufrufen. Eine Funktion, die Sie aufrufen, hat neben den Funktionsnamen mindestens noch den Funktionsaufruf-Operator `()`, der immer hinter dem Funktionsnamen notiert wird:

```
// einfacher Funktionsaufruf ohne Parameter
func();
```

In diesem einfachen Beispiel wird die Funktion ohne irgendwelche Parameter aufgerufen. Dies setzt natürlich voraus, dass sie auch ohne irgendwelche Parameter deklariert und definiert wurde. Das Ganze in der Praxis:

```
// func2.cpp
#include <iostream>
using namespace std;

// Funktions-Prototyp (Deklaration)
void func ( void );

int main(void) {
   cout << "Vor dem Funktionsaufruf\n";
   // Funktionsaufruf
   func();
   cout << "Nach dem Funktionsaufruf\n";
```

```
   return 0;
}

// Funktionsdefinition
void func ( void ) {
   cout << "func() ist aktiv\n";
}
```

Das Programm bei der Ausführung:

```
Vor dem Funktionsaufruf
func() ist aktiv
Nach dem Funktionsaufruf
```

Die hier verwendete Funktion `func()` hat weder einen Rückgabewert noch irgendwelche Parameter – weshalb hier auch `void` als Rückgabewert und Parameter angegeben wurde.

Hinweis [«]

Funktionen, die keinen Rückgabewert (also `void`) haben, werden in vielen anderen Programmiersprachen auch als *Prozeduren* bezeichnet. Dies gilt zwar nicht für C++, sollte aber trotzdem hier erwähnt werden.

Parameterübergabe an Funktionen (call by value)

Meistens werden Sie allerdings kaum »nackte« Funktionen ohne Parameter bzw. einen Rückgabewert schreiben. Abhängig davon, wie Sie die Funktion deklariert und definiert haben, können bzw. müssen Sie auch Parameter beim Aufrufen an die Funktion übergeben. Eine solche Parameterübergabe kann wie folgt aussehen:

```
// func3.cpp
#include <iostream>
using namespace std;

// Funktions-Prototyp (Deklaration)
void func ( int var );

int main(void) {
   int wert = 20;
   // Funktionsaufruf
   func( 10 );
   // Funktionsaufruf
   func ( wert );
   return 0;
}
```

```
// Funktionsdefinition
void func ( int var ) {
   cout << "func() : Wert von var = " << var << "\n";
}
```

Das Programm bei der Ausführung:

```
func() : Wert von var = 10
func() : Wert von var = 20
```

Anhand der Deklaration kann man schon erkennen, dass diese Funktion einen Parameter vom Typ `int` erwartet. Verwenden Sie keinen Parameter oder einen anderen Typ, meldet sich der Compiler mit einem Fehler. Aufrufen können Sie diese Funktion nun aus jeder anderen Funktion mit

```
func( Integer_Wert );
```

[»] Hinweis

In der Informatik bezeichnet man die beim Funktionsaufruf angegebenen Parameter als Aktualparameter. Die Parameter, die bei der Deklaration angegeben wurden, werden als Formalparameter bezeichnet.

Wenn Sie die Funktion aufgerufen haben, wird hierfür ein Stack-Frame (Stack-Rahmen = dynamischer Speicher) angelegt. In diesem Bereich wird Speicher für die einzelnen Parameter reserviert, die die Funktionen enthalten.

Der Parameter, den Sie der Funktion als Argument übergeben haben, wird hierbei auch gleich initialisiert (alles im Stack-Rahmen wohlgemerkt). Damit steht der Wert, den Sie der Funktion beim Aufruf mit übergeben haben, als Kopie in der Funktion zur Verfügung. Man spricht hierbei von einer *call by value*-Übergabe – dies ist also immer eine Kopie vom Original.

[»] Hinweis

Neben *call by value* existiert auch *call by reference*, womit statt eines Wertes eine Adresse kopiert wird. Diese Art des Aufrufs wird noch im Zusammenhang mit Zeigern näher besprochen.

Da also formale Parameter als Kopie der aktuellen Parameter fungieren, können Sie diese beliebig verändern, ohne dass dies Auswirkungen auf die Aktualparameter hat. Selbst wenn Sie beim Funktionsaufruf Variablen mit dem gleichen Namen verwenden würden wie die formalen Parameter, hat dies keine Auswirkung auf die Werte der aktuellen Parameter. Hierfür ein Beispiel:

```
// func4.cpp
#include <iostream>
using namespace std;
```

```
// Funktions-Prototyp (Deklaration)
void func ( int var );

int main(void) {
   int var = 20;
   // ... vor dem Funktionsaufruf
   cout << "main() : Wert von var = " << var << "\n";
   // Funktionsaufruf - call by value
   func ( var );
   // ... nach dem Funktionsaufruf
   cout << "main() : Wert von var = " << var << "\n";
   return 0;
}

// Funktionsdefinition
void func ( int var ) {
   var = var*2;
   cout << "func() : Wert von var = " << var << "\n";
}
```

Das Programm bei der Ausführung:

```
main() : Wert von var = 20
func() : Wert von var = 40
main() : Wert von var = 20
```

Dieses Beispiel zeigt recht gut, dass beim Aufruf von `func()` jeweils eine Kopie von `var` und ein Original von `var` in der `main`-Funktion existiert. Beide haben zwar denselben Namen, aber unterschiedliche Adressen. Natürlich bedeutet dies auch, dass die Kopie der Variablen in `func()` genauso lokal ist wie die lokale Variable in der `main`-Funktion. Beide Variablen sind nur innerhalb ihres Anweisungsblocks gültig und verwendbar.

Natürlich können Sie auch mehrere Parameter (auch unterschiedlichen Typs) in einer Funktion verwenden. Wichtig ist nur, dass die Aufruf-Reihenfolge der Parameter mit der Deklaration und Definition der Funktion übereinstimmt.

Rückgabewert von Funktionen

Wenn Sie Funktionen schreiben, die ein Teilproblem lösen, werden Sie wohl einen Wert an den Aufrufer zurückgeben wollen, um mit diesem Wert (gegebenenfalls mit weiteren Funktionen) weiterarbeiten zu können. Welchen Wert und von welchem Typ eine Funktion etwas zurückgibt, geben Sie ja bereits bei der Deklaration und Definition der Funktion vor, zum Beispiel:

```
int area ( int l, int b );
```

Hier haben Sie eine Funktion, die zwei Parameter vom Typ `int` erwartet und als Rückgabewert ebenfalls einen Integer-Wert zurückgibt. Um einen Wert aus einer Funktion an den Aufrufer zurückzugeben, müssen Sie die `return`-Anweisung verwenden. Mit der Angabe von

```
return n;
```

beenden Sie sofort die Funktion, egal, was sich dahinter noch für Code befindet. Der Rückgabewert, den Sie an die aufrufende Funktion zurückgeben, wurde hier mit `n` angegeben. Wollen Sie mehr als einen Wert aus einer Funktion zurückgeben, können Sie Strukturen (`struct`) verwenden, worauf in Abschnitt 2.8.1, »Strukturen«, eingegangen wird.

[»] **Hinweis**

`return` können Sie auch ohne weitere Parameter bei Funktionen verwenden, die keinen Rückgabewert zurückgeben (`void`). Hierbei wird die Funktion an der Position beendet, an der `return` aufgerufen wird.

[»] **Hinweis**

Verwenden Sie `return` in der `main`-Funktion, bedeutet dies somit auch das Ende des Programms – aber dazu weiter unten mehr.

Wollen Sie, dass der Aufrufer diesen Wert speichert, benötigen Sie eine Variable mit entsprechendem Typ und weisen dieser Variablen den Rückgabewert der Funktion mit dem Zuweisungsoperator = zu:

```
int ret;
...
ret = area ( 10, 20 );
```

Die Variable `ret` erhält nun den Rückgabewert, den die Berechnung der Funktion `area()` zurückgibt. Dies bedeutet allerdings nicht, dass der Rückgabewert einer Variablen zwingend benötigt wird. Ein Beispiel dazu:

```
cout << area(10, 20);
```

Hierzu wieder ein Listing, das zum besseren Verständnis beitragen soll:

```
// func5.cpp
#include <iostream>
#include <cstdlib>
using namespace std;
```

```
// Funktions-Prototyp (Deklaration)
int area ( int l, int b );

int main(void) {
   int var1, var2, ret;
   cout << "Bitte die Länge angeben  : ";
   if ( !(cin >> var1) ) {
      cerr << "Fehler bei der Eingabe!\n";
      exit(1);
   }
   cout << "Bitte die Breite angeben : ";
   if ( !(cin >> var2) ) {
      cerr << "Fehler bei der Eingabe!\n";
      exit(1);
   }
   ret = area ( var1, var2 );
   cout << "Die Fläche beträgt " << ret << "\n";
   return 0;
}

// Funktionsdefinition
int area ( int l, int b ) {
   int flaeche;
   flaeche = l * b;
   return flaeche;
}
```

Das Programm bei der Ausführung:

```
Bitte die Länge angeben  : 12
Bitte die Breite angeben : 11
Die Fläche beträgt 132
```

Die Funktion `area` könnten Sie noch um eine lokale Variable »erleichtern«, indem Sie die Berechnung gleich in die `return`-Anweisung verpacken:

```
int area ( int l, int b ) {
   return ( l * b );
}
```

In der Praxis sollte man noch eine Funktion einbauen, die überprüft, ob einer der Werte gleich 0 war. In solch einem Fall lohnt sich keine weitere Berechnung mehr. Hier ein Beispiel, wie dies in der Praxis aussehen kann:

```
// func6.cpp
#include <iostream>
```

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

// Funktions-Prototyp (Deklaration)
int area ( int l, int b );
bool not_null( int var );

int main(void) {
   int var1, var2, ret;
   cout << "Bitte die Länge angeben  : ";
   if ( ! (cin >> var1) ) {
      cerr << "Fehler bei der Eingabe!\n";
      exit(1);
   }
   cout << "Bitte die Breite angeben : ";
   if ( !(cin >> var2) ) {
      cerr << "Fehler bei der Eingabe!\n";
      exit(1);
   }
   if( not_null( var1 ) && not_null( var2 ) ) {
      ret = area ( var1, var2 );
   }
   else {
      cout << "Eine der Angaben ist 0!\n";
      exit(1);
   }
   cout << "Die Fläche beträgt " << ret << "\n";
   return 0;
}

// Funktionsdefinition
int area ( int l, int b ) {
   return (l * b);
}

bool not_null( int var ) {
   if ( var > 0 ) {
      return true;
   }
   else {
      return false;
   }
}
```

Sie sollten grundsätzlich eine Funktion schreiben, mit der Sie Daten auf ihre Richtigkeit überprüfen. Ganz besonders dann, wenn die Daten vom Anwender eingegeben werden. Man darf sich einfach niemals darauf verlassen, dass der User schon das Richtige eingeben wird.

Auch hier könnten Sie die Funktion `not_null` auf eine Codezeile begrenzen, indem Sie die Auswertung, ob der Wert der Variablen größer als 0 ist, gleich wieder in die `return`-Anweisung verpacken.

```
bool not_null( int var ) {
   return ( var > 0 );
}
```

Achten Sie aber darauf, dass bei all den Optimierungen und Kürzungen, die Sie am Quellcode vornehmen, die Lesbarkeit und vor allem die Verständlichkeit des Codes erhalten bleiben.

1.10.3 Lokale und globale Variablen

Wie Sie bereits im Beispiel des Listings *func5.cpp* bei der Funktion `area()` gesehen haben, kann man einer Funktion nicht nur Variablen übergeben, sondern auch Variablen darin deklarieren.

```
int area ( int l, int b ) {
   int flaeche;
   flaeche = l * b;
   return flaeche;
}
```

Bei dieser Variablen handelt es sich um eine lokale Variable. Das heißt, sie ist nur innerhalb des Funktionsrumpfs gültig.

Es ist auch möglich, Variablen global zu deklarieren. Eine globale Variable hat daher auch einen globalen Gültigkeitsbereich und kann von jeder Funktion – auch der `main`-Funktion – verwendet werden.

In der Praxis sollte man allerdings eine Variable so lokal wie möglich und so global wie nötig deklarieren, denn globale Variablen bringen leider auch Probleme in Bezug auf die Übersichtlichkeit des Codes mit sich. Wenn Sie eine Variable global angelegt haben und bei einem mehrere tausend Zeilen langen Code darauf zurückgreifen, erscheint das Ganze nicht mehr so übersichtlich, besonders wenn jetzt auch noch ein Dritter diesen Code überarbeiten muss.

Eine Frage liegt Ihnen aber sicherlich am Herzen: Was ist, wenn Sie eine globale und eine lokale Variable mit demselben Namen verwenden? Welche Variable erhält den Zuschlag? Hier gilt, dass bei gleichnamigen Variablen immer die

lokalste Variable den Zuschlag vom Programm erhält – wie das folgende Beispiel demonstrieren soll:

```
// func7.cpp
#include <iostream>
using namespace std;

// Globale Variable
int var = 123;

// Funktions-Prototyp (Deklaration)
void func1 ( int var );
void func2 ( void );

int main(void) {
   int var = 321;
   cout << "main()  : var = " << var << "\n";
   // ... per Parameter
   func1 ( var );
   func2 ( );
   return 0;
}

// Funktionsdefinition
void func1 ( int var ) {
   cout << "func1() : var = " << var << "\n";
}

void func2 ( void ) {
   cout << "func2() : var = " << var << "\n";
}
```

Das Programm bei der Ausführung:

```
main()  : var = 321
func1() : var = 321
func2() : var = 123
```

1.10.4 Standardparameter

Bei der Deklaration der Prototypen einer Funktion ist es in C++ (nicht in C) auch möglich, die Funktionsparameter mit einem bestimmten Wert zu belegen. Man spricht dabei auch von Standardparameter (Default-Parameter). Hat man beispielsweise einen Prototyp wie

```
void func ( int var );
```

deklariert, kann man den Parameter `var` folgendermaßen mit einem Standardwert belegen:

```
void func ( int var = 66 );
```

Wird jetzt beim Funktionsaufruf kein Argument mit übergeben, so wird der Standardwert `66` vom Compiler verwendet. Andernfalls wird – wie sonst auch – der Wert verwendet, der der Funktion als Argument übergeben wurde. Hierzu folgendes Beispiel:

```
// func8.cpp
#include <iostream>
using namespace std;

// Funktions-Prototyp mit Standardparameter
void func ( int var = 66 );

int main(void) {
   // Funktion mit dem Argument 99 aufrufen
   func ( 99 );
   // Funktion ohne Argument aufrufen
   func ( );
   return 0;
}

// Funktionsdefinition
void func ( int var ) {
   cout << "func1() : var = " << var << "\n";
}
```

Das Programm bei der Ausführung:

```
func1() : var = 99
func1() : var = 66
```

Um keine Missverständnisse aufkommen zu lassen: Der Name des Prototyps muss nicht mit dem Namen der Funktionsdefinition übereinstimmen, da die Standardzuweisung nach Position und nicht nach Namen vorgenommen wird. Der Prototyp könnte demnach auch wie folgt aussehen:

```
void func ( int = 66 );
```

Natürlich lässt sich dies auch mit mehreren Parametern machen – allerdings müssen Sie dabei beachten, dass die Zuordnung der Parameter von links nach rechts erfolgt. Wenn praktisch ein Parameter keinen Standardwert besitzt, kann auch der vorherige Parameter (links davon) keinen Standardwert haben. Am besten demonstriert man dies an Prototypen mit drei Parametern:

```
void func ( int par1 = 11, int par2 = 22, int par3 = 33 );
```

Es stehen Ihnen nun vier verschiedene Möglichkeiten zur Verfügung, diese Funktion aufzurufen:

```
// func9.cpp
#include <iostream>
using namespace std;

// Funktions-Prototyp mit Standardparameter
void func ( int par1 = 11, int par2 = 22, int par3 = 33 );

int main(void) {
   // Funktion ohne Argument aufrufen
   func ( );
   // Funktion mit einem Argument aufrufen
   func ( 44 );
   // Funktion mit zwei Argumenten aufrufen
   func ( 44, 55 );
   // Funktion mit drei Argumenten aufrufen
   func ( 44, 55, 66 );
   return 0;
}

// Funktionsdefinition
void func ( int par1, int par2, int par3 ) {
   cout << "(par1/par2/par3) = "
        << par1 << ' ' << par2 << ' '
        << par3 << ' ' << '\n';
}
```

Das Programm bei der Ausführung:

```
(par1/par2/par3) = 11 22 33
(par1/par2/par3) = 44 22 33
(par1/par2/par3) = 44 55 33
(par1/par2/par3) = 44 55 66
```

Anders sieht dies schon bei folgendem Prototyp aus:

```
void func ( int par1, int par2 = 22, int par3 = 33 );
```

Hier stehen Ihnen nur noch drei verschiedene Möglichkeiten zur Verfügung, da immer mindestens ein Argument für den ersten Parameter `par1` angegeben werden muss (und immer daran denken, dass die Zuordnung der Parameter von links nach rechts erfolgt). Die möglichen Funktionsaufrufe hierfür lauten:

```
   // Funktion mit einem Argument aufrufen
   func ( 44 );
   // Funktion mit zwei Argumenten aufrufen
   func ( 44, 55 );
   // Funktion mit drei Argumenten aufrufen
   func ( 44, 55, 66 );
```

Verwenden Sie hingegen nur einen Standardparameter, so haben Sie nur noch zwei mögliche Aufrufe dieser Funktion, da hier immer Argumente für die Parameter `par1` und `par2` benötigt werden.

```
// Prototyp mit einem Standardwert
void func ( int par1, int par2, int par3 = 33 );

...
   // Funktion mit zwei Argumenten aufrufen
   func ( 44, 55 );
   // Funktion mit drei Argumenten aufrufen
   func ( 44, 55, 66 );
...
```

Somit sind die folgenden drei Prototypen falsch, weil hier der vorangegangene Parameter keinen Standardwert enthält und der Compiler die Auswertung von links nach rechts ausführt.

```
// dreimal falsche Prototypen
void func ( int par1 = 11, int par2, int par3 );
void func ( int par1 = 11, int par2, int par3 = 33 );
void func ( int par1, int par2 = 22, int par3 );
```

1.10.5 Funktionen überladen

In C++ können Sie auch mehrere Funktionen mit dem gleichen Namen verwenden. Dabei können sich die Funktionen sowohl durch unterschiedliche Parametertypen als auch durch eine unterschiedliche Anzahl von Parametern unterscheiden. Dieser Vorgang wird als *Überladen* von Funktionen bezeichnet (oder auch als *Funktionspolymorphie*). So können Sie z. B. mehrere Funktionen mit dem Namen `do_stuff()` vereinbaren:

```
int do_stuff ( int par );
int do_stuff ( double par ) ;
```

Beide Prototypen haben zwar den gleichen Namen `do_stuff()`, unterscheiden sich aber durch unterschiedliche Datentypen als Parameter. Solange die Funktionsparameter unterschiedlich sind (und nur dann), können Sie auch den Rückgabewert verändern:

```
int do_stuff ( int par );
double do_stuff ( double par ) ;
```

Wie bereits erwähnt, können Sie auch eine unterschiedliche Anzahl von Funktionsparametern verwenden:

```
int do_stuff ( int par );
int do_stuff ( int par1, int par2 );
double do_stuff ( double par ) ;
double do_stuff ( double par1, double par2 );
```

Dass dies überhaupt funktioniert, liegt daran, dass der Compiler die Funktionen nicht anhand ihrer Namen identifiziert, sondern anhand ihrer Signatur. Die Signatur entsteht bei der Deklaration der Funktion aus einer Kombination der Funktionsnamen und der Parameterliste. Neben dem Überladen von Funktionen ist die Signatur auch beim Überschreiben von virtuellen Funktionen sehr wichtig. Durch diese Signatur sind allerdings die überladenen Funktionen für den Compiler nichts anderes als die üblichen Funktionen, wie Sie diese bisher auch verwendet haben.

Dank dieser Technik können Sie Funktionen mit ähnlichen Aktionen gleiche Namen geben. Dies ist auch beim Vereinbaren eines eigenen Datentyps sehr nützlich.

Die Technik, die der Compiler verwendet, um die richtige Funktion zu finden, ist in der Tat beeindruckend. Hier durchläuft der Compiler der Reihe nach folgende fünf Schritte:

- Der Compiler findet eine Funktion, die vollständig auf die Signatur passt, und muss keinerlei Improvisationen vornehmen.
- Der Compiler versucht eine Funktion zu finden, mit der sich eine integrale Promotion durchführen lässt, beispielsweise wird aus `bool` (`true`=1 und `false`=0) `int`.
- Jetzt versucht der Compiler, eine Standard-Typumwandlung durchzuführen, um eine entsprechende Funktion zu finden.
- Klappt es nicht mit einer Standard-Typumwandlung, sucht der Compiler nach einer benutzerdefinierten Typumwandlung für den Funktionsaufruf.
- Zum Schluss sucht der Compiler noch nach einer Funktion mit den drei Punkten (. . .) als Parameter – auch Ellipse genannt. Mit solchen Funktionen können theoretisch beliebig viele Parameter verwendet werden. In der Praxis verwende ich diese letzte Form der Suche nach einer Funktion gerne als den `else`-Zweig der Funktionsüberladung, um eine Fehlermeldung auszugeben.

Hinweis [«]

Wenn Sie interessiert, wie man eine variabel lange Argumentenliste für Funktionen auswerten kann, möchte ich Sie auf mein Buch »C von A bis Z« hinweisen. Dieses Buch finden Sie als Openbook unter *http://www.galileocomputing.de/* oder auf der Buch-CD.

Hier das Prinzip der Funktionsüberladung in der Praxis:

```
// func10.cpp
#include <iostream>
using namespace std;

// Funktions-Prototyp
int do_stuff ( int par );
int do_stuff ( int par1, int par2 );
double do_stuff ( double par );
double do_stuff ( double par1, double par2 );
void do_stuff ( ... );

int main(void) {
   int var1;
   double var2;
   // Aufruf von "int do_stuff (int)"
   var1 = do_stuff ( 10 );
   cout << var1 << '\n';
   // Aufruf von "double do_stuff (double)"
   var2 = do_stuff ( 10.2 );
   cout << var2 << '\n';
   // Aufruf von "int do_stuff (int, int)
   var1 = do_stuff ( 10, 11 );
   cout << var1 << '\n';
   // Aufruf von "double do_stuff(double, double)"
   var2 = do_stuff ( 10.11, 11.22 );
   cout << var2 << '\n';
   // Falscher Funktionsaufruf, daher ->
   // -> Aufruf von "void do_stuff(...)"
   do_stuff ( 'a', 'b', 'c' );
   return 0;
}

// Funktionsdefinition
int do_stuff ( int par ) {
   return (par * 2);
}
```

```
int do_stuff ( int par1, int par2 ) {
   return (par1 * par2);
}

double do_stuff ( double par ) {
   return (par * 2);
}

double do_stuff ( double par1, double par2 ) {
   return (par1 * par2);
}

void do_stuff ( ... ) {
   cout << "Fehler: Falscher Funktionsaufruf\n";
}
```

Das Programm bei der Ausführung:

```
20
20.4
110
113.434
Fehler: Falscher Funktionsaufruf
```

Das Überladen von Funktionen ist sicherlich eine tolle Sache in C++, aber Sie sollten bedenken, dass Sie häufig denselben Effekt mit vorbelegten Standardwerten erreichen. Da der Compiler ja auch die Standard-Typumwandlung übernimmt, können Sie das Beispiel oben (*func10.cpp*) bis auf zwei Funktionen kürzen – oder eigentlich bis auf eine, die zweite Funktion mit der Ellipse muss ja nicht unbedingt sein (Stilfrage). Hierzu das gleiche Beispiel nochmals, nur unter Verwendung von Standardparametern:

```
// func11.cpp
#include <iostream>
using namespace std;

// Funktions-Prototyp
double do_stuff ( double par1, double par2 = 2.0 );
void do_stuff ( ... );

int main(void) {
   int var1;
   double var2;
   // Aufruf von "int do_stuff (int)"
   var1 = do_stuff ( 10 );
```

```
   cout << var1 << '\n';
   // Aufruf von "double do_stuff (double)"
   var2 = do_stuff ( 10.2 );
   cout << var2 << '\n';
   // Aufruf von "int do_stuff (int, int)
   var1 = do_stuff ( 10, 11 );
   cout << var1 << '\n';
   // Aufruf von "double do_stuff(double, double)"
   var2 = do_stuff ( 10.11, 11.22 );
   cout << var2 << '\n';
   // Falscher Funktionsaufruf, daher ->
   // -> Aufruf von "void do_stuff(...)"
   do_stuff ( 'a', 'b', 'c' );
   return 0;
}

double do_stuff ( double par1, double par2 ) {
   return (par1 * par2);
}

void do_stuff ( ... ) {
   cout << "Fehler: Falscher Funktionsaufruf\n";
}
```

Ich denke, der Vorteil liegt auf der Hand. Wollen Sie zum Beispiel die Funktion `do_stuff()` verändern bzw. erweitern, müssen Sie dies nur einmal – statt wie im Beispiel zuvor viermal – tun. Sofern sich also die einzelnen Parameter nicht grundlegend voneinander unterscheiden, sollten Sie immer die Standardparameter der Funktionsüberladung vorziehen.

1.10.6 Inline-Funktionen

Beim Aufruf von Funktionen wird immer ein gewisser zusätzlicher Rechenaufwand benötigt. Ohne zu sehr ins Detail zu gehen, soll dies hier kurz erläutert werden.

Wenn ein Programm eine Funktion aufruft, wird (wie bereits erwähnt) ein sogenannter Stack-Rahmen (Stack-Frame) eingerichtet. Der Stack ist eine zusätzliche Speichereinheit, die das Programm für die aufgerufene Funktion verwendet. Der genaue Ablauf ist zwar architekturspezifisch, aber im Grunde gehen alle Architekturen ähnlich vor.

Zunächst wird die Rücksprungadresse auf dem Stack abgelegt, damit das Programm weiß, wohin es nach der Ausführung der Funktion zurückkehren kann.

Dann wird für den Rückgabetyp Platz auf dem Stack geschaffen. Lautet beispielsweise die Deklaration `int do_stuff( ..... )`, so wird Speicher für den Datentyp `int` reserviert. Neben dem Rückgabetyp wird selbstverständlich auch Speicher für die einzelnen Parameter der Funktion (falls verwendet) auf dem Stack bereitgestellt. Erst jetzt wird gewöhnlich die Funktion ausgeführt und somit auch die Definition. Das bedeutet, dass noch weiterer Platz auf dem Stack für die Definition der lokalen Variablen der Funktion benötigt wird.

Für viele einfache Funktionen, die Sie bisher geschrieben haben, stellt sich die Frage, ob sich der Aufwand lohnt oder man nicht gleich die paar Zeilen in die `main`-Funktion schiebt?

Für solche Zwecke wurden in C++ die `inline`-Funktionen geschaffen. Stellen Sie das Schlüsselwort `inline` vor eine Funktion, so wird der Compiler angewiesen, diese Funktion nicht als eine aufrufbare Funktion in den Maschinencode zu übersetzen, sondern den Code an der Stelle zu setzen, an der die Funktion aufgerufen wird. Damit sollte der Aufwand, der bei einem Funktionsaufruf betrieben wird, vermieden werden. Beispielsweise sei folgender Code gegeben:

```
// func12.cpp
#include <iostream>
using namespace std;

// Funktions-Prototyp
inline int max_int ( int a, int b );

int main(void) {
   int var1 = 100, var2 = 200, max;
   max = max_int ( var1, var2 );
   cout << max << " ist der größere Wert\n";
   return 0;
}

// Funktionsdefinition
inline int max_int ( int a, int b ) {
   if ( a >= b ) {
      return ( a );
   }
   else {
      return ( b );
   }
}
```

Der `inline`-Theorie nach würde der Compiler aus diesem Quellcode nun Folgendes machen:

```
// func12.cpp
#include <iostream>
using namespace std;

int main(void) {
   int var1 = 100, var2 = 200, max;
   if ( var1 >= var2 ) {
      max = var1;
   }
   else {
      max = var2;
   }
   cout << max << " ist der größere Wert\n";
   return 0;
}
```

Ob der Compiler dies in der Praxis allerdings auch so macht, wie Sie wollen, entscheidet er doch letztendlich selbst. Wenn Sie eine Funktion mit `inline` notieren, schlagen Sie dem Compiler vor, diese als `inline` zu verwenden. Eine Garantie gibt es dafür allerdings nicht. Dennoch haben Sie sehr gute Chancen, dass die Funktion als `inline` vom Compiler nominiert wird, wenn der Code der Funktion recht gering ist.

Hinweis [«]

Sofern Sie eine Funktion als `inline` notieren und diese im Programm häufig an unterschiedlichen Stellen aufrufen, müssen Sie beachten, dass sich dadurch natürlich auch der Umfang des Maschinencodes erhöhen kann.

Am sinnvollsten setzt man `inline`-Funktionen in Schleifen ein, die sehr häufig ein und dieselbe Funktion aufrufen. Richtig eingesetzt lässt sich damit die Laufzeit verbessern und der Objektcode gegebenenfalls reduzieren.

Hinweis [«]

Mittlerweile sind die Compiler so klug, dass sie solche `inline`-Optimierungen selbst durchführen, auch wenn Sie eine Funktion vielleicht gar nicht als `inline` notiert haben.

Bei all der Hysterie um das Thema »Optimierung« sollte man nie das Ziel aus den Augen verlieren. Optimierungen sollten immer nur dann durchgeführt werden, wenn es bei der Ausführung des Programms zu Engpässen (z. B. Wartezeiten) kommt.

C-Programmierer sollten nicht den Fehler machen, die `inline`-Funktionen mit den `define`-Makros zu vergleichen. Die `define`-Makros werden vom Präprozessor expandiert, die `inline`-Funktionen vom Compiler. Das bedeutet, dass bei den

C-Makros keine Typüberprüfung stattfindet. Genau aus diesem Grund wurden ja die `inline`-Funktionen eingeführt. Mittlerweile steht auch dem C-Programmierer seit dem C99-Standard `inline` zur Verfügung.

1.10.7 Rekursionen

Rekursionen sind Funktionen, die sich immer wieder selbst aufrufen und somit neu definieren, bis eine bestimmte Abbruchbedingung eintrifft. Fehlt diese Abbruchbedingung, wird sich die Rekursion ähnlich wie eine ungewollte Endlosschleife verhalten und den Rechner eventuell stark belasten. Schließlich fordert jeder Funktionsaufruf Platz auf dem Stack (Rücksprungadresse sichern; Rückgabewert; Platz für die Funktionsparameter; lokale Variablen usw.). Da hierbei keine Funktion an den Aufrufer zurückkehrt – bis die Abbruchbedingung gültig ist –, kann es passieren, dass der zur Verfügung stehende Stack für das Programm voll- bzw. überläuft (Stack Overflow).

Ein klassisches Beispiel dafür dürfte die Berechnung der Fakultät einer Zahl *n* sein. So lautet zum Beispiel die Fakultät der Zahl »6« 720 (errechnet aus `1*2*3*4*5*6`). Somit kann man so lange eine bestimmte Zahl um eins dekrementieren, bis die Abbruchbedingung 0 erreicht wurde. Das Beispiel dazu:

```
// func13.cpp
#include <iostream>
using namespace std;

// Funktions-Prototyp
long fakul( long n );

int main(void) {
   long val;
   val = fakul( 6 );
   cout << "Fakultät aus 6 ist " << val << '\n';
   val = fakul( 10 );
   cout << "Fakultät aus 10 ist " << val << '\n';
   return 0;
}

// Funktions-Definition
long fakul( long n ) {
   if( n ) {
      return n * fakul(n-1);
   }
   return 1;
}
```

Das Programm bei der Ausführung:

```
Fakultät aus 6 ist 720
Fakultät aus 10 ist 3628800
```

Die Funktion `fakul()` ruft sich so lange mit `n*n-1` selbst auf, bis `n` gleich `0` ist. Hierbei könnte man auf `n*1` auch verzichten, weil sich das Ergebnis nicht mehr verändern wird.

Das hört sich in der Theorie ganz gut an, aber in der Praxis ist man mit der iterativen Problemlösung immer noch wesentlich effizienter. Hierzu die iterative Lösung zur Fakultät einer Zahl:

```
long fakul(int n) {
   int x = n;
   while(--x) {
      n *= x;
   }
   return n;
}
```

Diese Funktion erfüllt denselben Zweck und ist auch noch einfacher und verständlicher. Wozu also Rekursionen verwenden, wenn zum einen der Code dadurch häufig schwerer zu verstehen ist und zum anderen der Aufwand mit dem Stack enorm sein kann? Es gibt durchaus Dinge, die man mit Rekursionen einfacher und schneller lösen kann (etwa binäre Bäume, Sortierroutinen wie Quicksort usw.), trotzdem gibt es häufig auch eine iterative Lösung. Um sich allerdings mit dem Pro und Contra auseinanderzusetzen, müsste man sich schon mehr mit den »Algorithmen« befassen – aber das Thema ist schon sehr spezifisch und vor allem umfangreich, weshalb man hier bei Bedarf mit spezieller Literatur gut beraten ist.

Hinweis **[«]**

Einen umfassenderen Einblick in die Themen »Rekursionen« und »Algorithmen« erhalten Sie in meinem Buch »C von A bis Z«, das Sie als Openbook unter *http://www.pronix.de/* oder auf der Buch-CD finden.

1.10.8 Die »main«-Funktion

Nach dem C++-Standard muss jedes ausführbare Programm maximal eine Funktion mit dem Namen `main()` besitzen. Diese Funktion ist stets die erste Funktion, die beim Programmstart ausgeführt wird.

Ebenfalls fordern nach dem C++-Standard die `main`-Funktionen den Rückgabewert `int`. Zwar findet man in (zumeist älteren) Büchern immer noch folgende Schreibweise:

```
void main(void) {
   // Anweisungen
}
```

Aber nach neuem Standard ist dies nicht mehr richtig. Der Compiler dürfte sich bei dieser Verwendung sowieso bei Ihnen beschweren. Richtig ist also immer:

```
int main(void) {
   // Anweisungen
   return 0;
}
```

Weiterhin ist auch eine Variante mit zwei Parametern erlaubt:

```
int main(int argc, char **argv) {
   return 0;
}
```

Damit können Sie der `main`-Funktion beim Programmstart auch einige Argumente mitgeben. Die Namen der Bezeichner sind hier zwar frei wählbar, aber in der Praxis werden gewöhnlich die Namen `argc` und `argv` verwendet.

Über den Rückgabewert der `main`-Funktionen sind schon regelrechte »Flamewars« ausgebrochen. Generell ist dieser Wert abhängig von der Umgebung des Betriebssystems. Unter Linux/UNIX bedeutet ein Rückgabewert von `0`, dass ein Programm erfolgreich beendet wurde; alles andere bedeutet, dass etwas fehlgeschlagen ist. So können Sie unter Linux/UNIX mit

```
$you@host > echo $?
0
```

ermitteln, welchen Rückgabewert das zuletzt gestartete Programm (bzw. Kommando) zurückgegeben hat.

Andere Betriebssysteme können auch einen anderen Rückgabewert als erfolgreiche Beendigung erwarten, was bedeutet, dass es hierbei keinen »portablen« Standard gibt, aber der Konvention nach wird hierfür immer `0` verwendet.

1.11 Präprozessor-Direktiven

Bevor das C++-Programm vom Compiler übersetzt wird, wird der Präprozessor aktiv. Dieser fasst u. a. String-Literale zusammen, entfernt Zeilenumbrüche mit

vorgestelltem Backslash, löscht die Kommentare des Quelltexts sowie die Whitespace-Zeichen zwischen den Tokens. Zusammengefasst führt der Präprozessor rein textuelle Manipulationen am C++-Quelltext durch. Ist der Präprozessor fertig, erhält der Compiler den so angepassten Quelltext zum Übersetzen.

Diese für den Präprozesser gedachten Zeilen werden *Direktiven* (oder auch *Präprozessor-Direktiven*) genannt und beginnen immer mit dem Hash-Zeichen # am Anfang einer Zeile und enden am Zeilenende. Beachten Sie, dass im Gegensatz zu einer C++-Anweisung eine Direktive nicht mit einem Semikolon beendet wird. Fügen Sie hierbei ein Semikolon am Ende ein, kann dies zu unerwarteten Ergebnissen führen. Sollten Ihre Präprozessor-Direktiven länger werden, können Sie diese in der nächsten Zeile fortsetzen, wenn Sie einen Backslash an das Zeilenende setzen.

Da die Direktiven nicht vom Gültigkeitsbereich abhängig sind, können diese irgendwo im Quelltext stehen. Pro Zeile ist eine Direktive erlaubt.

Die Hauptanwendungsgebiete von Präprozessor-Direktiven sind das Einkopieren von Header- und/oder Quelldateien, das Einbinden symbolischer Konstanten sowie die bedingte Kompilierung.

1.11.1 Die »#define«-Direktive

Mit der `#define`-Direktive lassen sich einfache Makros realisieren. Die Syntax lautet:

```
#define makroname ersetzungsname
```

Mit dieser Direktive veranlassen Sie den Präprozessor, überall im Quelltext `Makroname` durch den Ersetzungstext `Ersetzungsname` zu ersetzen. In der Praxis schreibt man den Makronamen gewöhnlich groß. Ein einfaches Beispiel hierfür:

```
// define1.cpp
#include <iostream>
#define NUMBER 5
#define STRING "Hallo"
using namespace std;

int main(void) {
   for( int i = 0; i < NUMBER; i++ ) {
      cout << STRING << '\n';
   }
   return 0;
}
```

In dem Beispiel wurden zwei einfache Ersetzungsmakros `NUMBER` mit dem Wert `5` und `STRING` mit der Zeichenfolge **`"Hallo"`** definiert. Somit werden vom Präprozessor alle Makros mit entsprechenden Namen durch den entsprechenden Wert ersetzt. Der Compiler würde praktisch vereinfacht folgenden Quellcode zum Übersetzen erhalten:

```
// define1.cpp
#include <iostream>
using namespace std;

int main(void) {
   for( int i = 0; i < 5; i++ ) {
      cout << "Hallo" << "\n";
   }
   return 0;
}
```

[»] **Hinweis**

Dies ist natürlich sehr vereinfacht dargestellt, da nach dem Präprozessor-Lauf erheblich mehr zu finden ist, als man annehmen würde. Jeder Compiler bietet einen Schalter an, der Ihnen den Quelltext nach dem Präprozessor und vor dem Compiler-Lauf ausgibt (bei der GCC ist dies beispielsweise das Compiler-Flag `-E`).

Neben der Möglichkeit, einfache Makros zu definieren, können Sie auch parametrisierte Makros verwenden. Ein einfaches Beispiel dazu:

```
#define SQRE(x) ( (x) * (x))  // Quadrieren
```

Im vorliegenden Fall haben Sie den formalen Parameter `x`. Dieser kann auf der rechten Seite des Makros beliebig oft verwendet werden. Dabei muss beachtet werden, dass dieser formale Parameter ebenfalls auf der rechten Seite in Klammern stehen muss. Folgendes Beispiel demonstriert Ihnen, was passiert, wenn Sie keine Klammerung verwenden:

```
// define2.cpp
#include <iostream>
#define SQRE1(x) ( (x) * (x))  // Quadrieren
#define SQRE2(x) (x * x)       // Quadrieren
using namespace std;

int main(void) {
   int wert = 5;
   cout << "SQRE1(wert): " << SQRE1(wert) << '\n';
   cout << "SQRE2(wert): " << SQRE2(wert) << '\n';
   cout << "SQRE1(wert+1): " << SQRE1(wert+1) << '\n';
```

```
   cout << "SQRE2(wert+1): " << SQRE2(wert+1) << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
SQRE1(wert): 25
SQRE2(wert): 25
SQRE1(wert+1): 36
SQRE2(wert+1): 11
```

Der Präprozessor versteht nämlich keine C++-Syntax, und aus

```
SQRE2(x) (x * x)
```

wird

```
(wert+1 * wert+1)
```

und das Ergebnis ist nicht das gleiche wie

```
((wert+1) * (wert+1))
```

Natürlich können auch mehrere Argumente als Parameter eines Makros verwendet werden:

```
#define MAX(x, y) ( (x)<=(y) ?(y) :(x) )
```

Das Makro ermittelt den größeren Wert von `x` und `y`. Wie bei einer Funktion werden hierbei die einzelnen Argumente mit einem Komma getrennt. Und sollte sich mal ein Makro über mehrere Zeilen erstrecken, so wird die nächste Zeile auch noch als Makro betrachtet, wenn die vorherige Zeile mit einem Backslash beendet wurde:

```
// Zwei Werte tauschen
#define TAUSCHE(x, y)   { \
   int j; \
   j=x;   \
   x=y;   \
   y=j;   \
  }
```

»#define« oder »const«

Ganz klar, die `define`-Direktive ist unverzichtbar, wenn es um Dinge wie die bedingte Kompilierung geht, aber in der Praxis wird häufig `const` vorgezogen. Für einen Einsteiger in die Programmiersprache C++ ist der Unterschied zunächst nicht ganz einsichtig, aber mit fortschreitenden Kenntnissen werden die folgenden Punkte deutlicher erscheinen:

- C++ überprüft die Syntax einer `const`-Anweisung sofort. Bei `#define` findet erst eine Überprüfung statt, wenn das Makro verwendet wird.
- Eine `const`-Anweisung kann auf so ziemlich jeden C++-Typ definiert werden (also auch auf Klassen und Strukturen). `#define` hingegen beschränkt sich nur auf einfache Konstanten.
- `const` verwendet die üblichen C++-Regeln für die Gültigkeitsbereiche (siehe Kapitel 3, »Gültigkeitsbereiche, spezielle Deklarationen und Typumwandlungen«). `#define`-Konstanten sind immer und überall gültig.
- `const` verwendet die C++-Syntax – `#define` hat eine eigene Syntax.

Inline-Funktionen oder parametrisierte Makros

Das parametrisierte Makro

```
#define SQRE2(x) ( x * x)      // Quadrieren
```

hat bereits gezeigt, dass solche Konstrukte unangenehme Nebeneffekte haben können. Daher werden auch hier in der Praxis meistens `inline`-Funktionen bevorzugt.

```
inline int SQRE( const int x ) {
   return ( x * x );
}
```

Jetzt hat man allerdings den Nachteil, dass man die `inline`-Funktion nur für den Datentyp `int` verwenden kann. Beim Makro hingegen war man nicht vom Datentyp abhängig. Man kann nun für jeden Datentyp eine `inline`-Funktion schreiben, oder aber man verwendet eines der besten Features von C++, den Funktions-Template (siehe Abschnitt 5.1, »Funktions-Templates«).

1.11.2 Die »#undef«-Direktive

Der Geltungsbereich von symbolischen Konstanten bzw. Makros reicht vom Punkt der Deklaration mit `#define` bis zur Aufhebung mit `#undef`. Die Aufhebung mit Hilfe von `#undef` ist aber optional. Wird `#undef` nicht verwendet, reicht der Geltungsbereich bis zum Dateiende.

Das folgende Listing wird sich nicht übersetzen lassen, weil beim Compiler-Lauf der zweite Makroname `WERT` nicht mehr definiert ist, da dieser zuvor mit `#undef` aufgehoben wurde.

```
// undef1.cpp
#include <iostream>
// Makronamen WERT definieren
```

```
#define WERT 5
using namespace std;


int main(void) {
   cout << WERT << "\n";
// Definierten Makronamen aufheben
#undef WERT
   // !!! Compilerfehler !!!
   cout << WERT << "\n";
   return 0;
}
```

1.11.3 Die »#include«-Direktive

Mit der Präprozessor-Direktive `include` kann ein Programm Quellcode aus einer anderen Datei einkopieren. Die Syntax lautet:

```
#include tokens
```

Diese Angabe bewirkt, dass die Zeile durch den kompletten Inhalt der in `tokens` angegebenen Datei ersetzt wird. Hat die Direktive die Form

```
#include <datei>
```

so wird gewöhnlich im *include*-Verzeichnis des Compilers bzw. in einem spezifizierten Verzeichnis nach dieser Datei gesucht. Auf UNIX-Systemen finden Sie dieses Verzeichnis gewöhnlich in */usr/include* oder */usr/local/include* – unter MS Windows ist dies abhängig vom Installationsverzeichnis des Compilers. Meistens handelt es sich bei der Angabe mit den spitzen Klammern um Standard-Header-Dateien wie beispielsweise:

```
#include <iostream>
```

Hier gleich ein Hinweis für Einsteiger in die Programmierung, da die Header-Dateien häufig mit Bibliotheken gleichgestellt werden: Standard-Header-Dateien wie `iostream` werden eingesetzt, um die von den Bibliotheksfunktionen verwendeten Datenstrukturen und Makros zu definieren.

Natürlich kann man auch eigene Header-Dateien schreiben, um beispielsweise Konstanten und Datenstrukturen darin zu speichern. Dies ist besonders hilfreich, wenn sich ein Programm über mehrere Dateien erstreckt – was gerade beim Programmieren im Team der Fall ist. Solche Angaben zu »lokalen« Header-Dateien werden zwischen doppelte Anführungsstriche gestellt:

```
#include "datei.h"
```

Natürlich können Sie auch einen relativen Pfad zur Datei angeben

```
#include "../../../datei.h"
```

oder aber auch einen absoluten Pfad

```
#include "/home/user/datei.h"
```

[»] **Hinweis**

Mittlerweile unterstützt auch MS Windows den normalen Schrägstrich als Verzeichnistrenner. Früher musste man immer den rückwärts gerichteten Schrägstrich alias Backslash dafür verwenden.

Gewöhnlich werden Header-Dateien für Definitionen, Deklarationen, Makros, Konstanten und `Inline`-Funktionen verwendet. Es ist aber möglich, Code in eine Header-Datei zu schreiben und zu verwenden. Bei einem guten Programmierstil würde man jedoch den Code in *cpp*-Dateien schreiben und den Rest in Header-Dateien. Natürlich ist es auf der anderen Seite auch möglich, *cpp*-Dateien von einer anderen Header-Datei aus einzubinden (mit `include`), was aber auch nicht unbedingt als »stilvolle« Programmierung gilt.

In den nächsten Abschnitten werden Sie ohnehin nur die Standard-Header-Dateien verwenden. Mehr darüber, wie Sie zum Beispiel eigene Header-Dateien erstellen können, werden Sie in Abschnitt 4.2.7, »Ein Programm organisieren«, erfahren.

1.11.4 Die Direktiven »#error« und »#pragma«

Mit der `#error`-Direktive können Sie den Übersetzungsvorgang des Compilers mit einer eigenen Fehlermeldung abbrechen. Dies ist sinnvoll, wenn zum Beispiel eine Funktion oder ein Programm nicht fertig ist oder man einen wichtigen Hinweis hinterlassen will. Ein einfaches Beispiel dazu:

```
// error1.cpp
#include <iostream>
#include "myownheader.h"
using namespace std;

// viele Deklarationen

int main(void) {
   // ... viele Anweisungen
   // ...
   #error "Bitte neue Headerdatei besorgen !"
   // ...
```

```
   return 0;
}

// viele Definitionen
```

Der Compiler beendet die Übersetzung mit der Fehlermeldung, dass man sich doch bitte die neue Header-Datei besorgen möge. Anschließend kann man diese Direktive auskommentieren oder gleich ganz entfernen.

`#pragma` sind compilerspezifische Direktiven und von Compiler zu Compiler verschieden. Wenn ein Compiler eine bestimmte `#pragma`-Direktive nicht kennt, wird diese ignoriert. Mit Hilfe dieser Pragmas können Compiler-Optionen definiert werden, ohne mit anderen Compilern in Konflikt zu geraten. Da das Verhalten von `#pragma`-Anweisungen stark systemabhängig ist, soll darauf nicht näher eingegangen werden. Welche Pragmas Ihr Compiler unterstützt, entnehmen Sie bitte dem C++-Manual Ihres Compilers.

1.11.5 Bedingte Kompilierung

Oft steht man vor dem Problem, einen Code schreiben zu müssen, der auf mehreren Betriebssystemen laufen soll. Zwar ist dies, dank Standard, heutzutage kein unlösbares Problem mehr, aber dennoch gibt es immer wieder kleinere Differenzen. Um jetzt nicht gleich für jedes Betriebssystem einen eigenen Code schreiben zu müssen, gibt es die Möglichkeit der bedingten Kompilierung.

Neben den unterschiedlichen Betriebssystemen gibt es außerdem noch unterschiedliche Compiler, die auch einige kleine Eigenheiten mitliefern. Auch hier kann man mit einer bedingten Kompilierung dafür sorgen, dass diese »Eigenheit« auch verwendet wird, wenn der Quellcode mit dem entsprechenden Compiler übersetzt wird.

Folgende drei Direktiven von Bedingungsanweisungen für den Präprozessor stehen Ihnen zur Verfügung:

```
#if Konstanter_Ausdruck
#ifdef Konstanter_Ausdruck
#ifndef Konstanter_Ausdruck
```

Mit der `#if`-Direktive überprüfen Sie, ob der Ausdruck (es muss ein Konstantenausdruck sein) einen Wert ungleich 0 zurückgibt. `#ifdef` überprüft, ob ein Bezeichner `Konstanter_Ausdruck` bereits definiert ist, und mit `#ifndef` wird geprüft, ob der Bezeichner `Konstanter_Ausdruck` dem Präprozessor nicht bekannt ist – also das Gegenteil von `#ifdef`. Bei den Bedingungen dürfen mit den logischen Operatoren `||` bzw. `&&` mehrere Ausdrücke miteinander verknüpft und somit überprüft werden:

```
#ifdef Ausdruck1 || Ausdruck2
...
```

Einer dieser drei Bedingungsanweisungen für den Präprozessor können nun beliebig viele Zeilen Code folgen und weitere Direktiven:

```
#elif Konstanter_Ausdruck
#else
#endif
```

Mit der Direktive `#elif` können beliebig viele weitere Ausdrücke überprüft werden. Eine `#else`-Direktive ist optional und kann am Ende von beliebig vielen `#elif`-Direktiven oder mindestens einer `#if`, `#ifdef` oder `#ifndef`-Direktive folgen. Am Ende muss immer eine `#endif`-Direktive stehen, wodurch der Präprozessor weiß, dass hier die bedingte Kompilierung zu Ende ist. In der Praxis sieht ein solches Konstrukt wie folgt aus:

```
#ifdef DIES
#   define FUNCTION(param) linux_func(param)
#elif JENES
#   define FUNCTION(param) dos_func(param)
#else
#   define FUNCTION(param) unknow_func(param)
#endif
```

Oder:

```
#if LANG == 1
#   include "german_header.h"
#elif LANG == 2
#   include "english_header.h"
#else
#   error "Es wird nur Englisch und Deutsch unterstützt"
#endif
```

Ein einfach ausführbares Beispiel in der Praxis sieht demnach wie folgt aus:

```
// ifdef1.cpp
#include <iostream>
using namespace std;

#ifdef __MSDOS__
// Programm läuft unter einem echten MS-DOS
// Hier kommt der Code dafür hin
#   define CODE "MS-DOS"
```

```
#elif __WIN32__ || _MSC_VER
// Programm läuft in einer Win32-Konsole oder
// wurde mit dem Microsoft-Compiler übersetzt
#   define CODE "Win32 oder MS-VC++\n"
#elif __unix__ || __linux__
// Programm wird unter einem Linux/UNIX-System
// übersetzt
#   define CODE "Linux/UNIX\n"
#else
// Präprozessor konnte keines der Systeme ausmachen ...
#define CODE "Unbekanntes System\n"
#endif

int main(void) {
   cout << CODE;
   return 0;
}
```

Je nachdem, auf welchem System das Programm übersetzt wird, erhält man eine entsprechende Ausgabe. Im Grunde ähnelt das Ganze den gewöhnlichen `if`-Abfragen von C++, nur für den Präprozessor.

Wie schon erwähnt, umfasst die bedingte Kompilierung neben maschinen-spezifischen Abfragen auch compilerspezifische. Daher folgt nun ein Überblick über die gängigen Compiler und Systeme und die entsprechenden Konstanten, die hierfür verwendet werden können. Zunächst die Konstanten für die Compiler:

Konstante	Compiler
`_MSC_VER`	Microsoft C ab Version 6.0
`_QC`	Microsoft Quick C ab Version 2.51
`__TURBOC__`	Borland Turbo C, Turbo C++ und BC++
`__BORLANDC__`	Borland C++
`__ZTC__`	Zortech C und C++
`__SC__`	Symantec C++
`__WATCOMC__`	WATCOM C
`__GNUC__`	Gnu C
`__EMX__`	Emx Gnu C

Tabelle 1.22 Konstanten für bestimmte Compiler

Und jetzt noch Konstanten, die das Betriebssystem betreffen:

Konstante	Betriebssystem
__unix__ oder __unix	UNIX-System
__MS_DOS__	MS-DOS
__WIN32__	Windows ab 95
__OS2__	OS2
_Windows	Zielsystem Windows
__NT__	Windows NT
__linux__	Linux
__FreeBSD__	FreeBSD
__OpenBSD__	OpenBSD
_SGI_SOURCE	SGI-IRIX mit Extension *.sgi
_MIPS_ISA	SGI-IRIX
__hpux	HP-UX

Tabelle 1.23 Konstanten für bestimmte Betriebssysteme

Die meisten Projekte bestehen gewöhnlich aus mehreren Header- und Codedateien. Alle diese Dateien werden beim Übersetzen zunächst zu Objektdateien kompiliert und anschließend von einem Linker zu einer ausführbaren Datei zusammengefasst. Nehmen wir einmal an, Sie haben zwei Header-Dateien, *file1.h* und *file2.h*, die die Header-Datei *myfile.h* inkludieren. Dann würde praktisch zweimal beim Präprozessor-Lauf die Datei *myfile.h* eingelesen. Damit würden Konstanten mehrmals definiert, was noch kein Fehler wäre; aber sobald eine Datenstruktur oder eine Union mehrmals definiert würde, ist dies definitiv ein Fehler.

Um dieses Problem zu lösen, braucht man nur in der entsprechenden Header-Datei – bei der die Gefahr besteht, dass sie mehrmals inkludiert wird – am Anfang folgenden Code einzufügen:

```
#ifndef MYFILE_H
#   define MYFILE_H
// Hier kommt der Inhalt der Code-Datei hin
#endif
```

In diesem Fall, wenn die Header-Datei *myfile.h* bereits eingebunden wurde, werden alle weiteren Definitionen bzw. der Code bis zur `#endif`-Direktive versteckt, so dass es nicht mehr zu Problemen kommen kann. Wurde die Header-Datei *myfile.h* noch nicht definiert (`#ifndef`), wird diese mit `#define` definiert und der entsprechende Code einkopiert.

In diesem Kapitel werden die höheren Datentypen beschrieben. Dabei handelt es sich um Typen, die aus den Basistypen gebildet werden. In C++ sind dies Zeiger, Arrays (oder auch Vektoren), Referenzen und Strukturen. Klassen gehören eigentlich auch in dieses Kapitel, aber damit ich hier nicht ständig vor- und zurückgreifen muss, werden die Klassen zu gegebener Zeit behandelt.

2 Höhere und fortgeschrittene Datentypen

2.1 Zeiger

Zeiger werden in C/C++ immer noch als das komplizierteste Thema hingestellt. Dabei sind Zeiger eigentlich gar nicht so schwer zu verstehen, wie dies auf den ersten Blick scheint – wohl stellen sie aber für einen Anfänger eine immer noch recht große Hürde dar. Das Problem beim Verständnis von Zeigern ist zunächst nicht die Funktionalität, sondern das »wozu«. Zudem muss hierbei erwähnt werden, dass sich der Aha-Effekt erst später einstellen wird. Hier geht es zunächst nur um die Grundlagen für die Zeiger.

Es wurde ja bereits erwähnt, dass alle in diesem Kapitel beschriebenen Typen aus den Basisdatentypen gebildet werden – also auch ein Zeiger. Alle Basisdatentypen müssen vom Rechner verwaltet werden. Dies geschieht, indem das System den Wert, die Größe und die Adresse im Arbeitsspeicher ablegt:

```
int var = 100;
```

Hiermit wird praktisch die Variable `var` mit dem Wert `100` und der Größe (abhängig von der Rechnerarchitektur) von vier Bytes im Arbeitsspeicher an einer bestimmten Adresse abgelegt. Im Arbeitsspeicher ergibt sich dadurch das in Abbildung 2.1 dargestellte Bild.

Bei der Adresse (hier `0000`) handelt es sich um eine erfundene Adresse. Welche Adresse `var` bei Ihnen im Arbeitsspeicher wirklich belegt, können Sie mit dem Adressoperator `&` vor der Variablen ermitteln:

```
// zeiger1.cpp
#include <iostream>
```

```
using namespace std;

int main(void) {
   int var = 100;
   cout << "Adresse von var : " << &var << '\n';
   return 0;
}
```

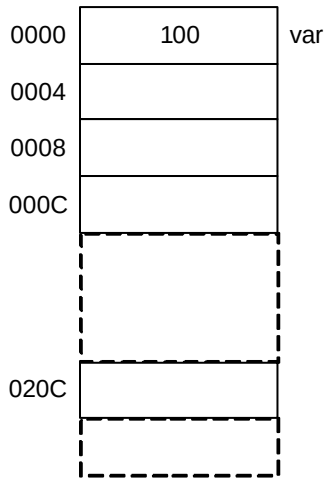

Abbildung 2.1 Speicherbelegung einer Variablen im Arbeitsspeicher

Das Programm bei der Ausführung:

```
Adresse von var : 0x22ff74
```

Beachten Sie aber, dass sich die Adressierungen vom Rechnertyp unterscheiden können. Wie viel Platz die Variable `var` im Arbeitsspeicher benötigt, geben Sie ja bereits bei der Deklaration der Variablen durch den Datentyp an.

[»] **Hinweis**

Der Compiler unterscheidet zwischen dem unären Adressoperator `&` und dem binären bitweisen UND `&`, da der unäre Adressoperator einen Operanden, der binäre hingegen zwei benötigt.

2.1.1 Zeiger deklarieren

Die Syntax der Deklaration eines Zeigers sieht wie folgt aus:

```
Typ *zeiger;
```

Der Typ des Zeigers muss immer vom selben Typ sein wie der Datentyp des Speicherplatzes, auf den der Zeiger verweist. Zeiger sind also typgebunden.

Hinweis [«]

Es sollte erwähnt werden, dass es mehr gibt als Zeiger auf einfache Speicherobjekte – wie hier am Anfang mit den Basisdatentypen gezeigt wird.

In der Praxis sieht diese Deklaration so aus:

```
// Deklaration von Zeiger auf char-Variable
char *cptr;
// Deklaration von Zeiger auf int-Variable
int *iptr;
// Deklaration von Zeiger auf double-Variable
double *dptr;
```

Sie können zur Deklaration eines Zeigers zwei Schreibweisen verwenden:

```
// Sternchen vor dem Zeiger-Namen
int *iptr1;
// Sternchen nach dem Typen-Namen
int* iptr2;
```

Welche Variante Sie auswählen, bleibt Ihnen überlassen. Wer zuvor bereits in C programmiert hat, wird sich mit der ersten Schreibweise recht schnell als C-Geübter outen, da diese vorwiegend in C eingesetzt wurde. Die meisten C++-Programmierer verwenden die zweite Variante (vielleicht, um sich nicht als alter C-Programmierer enttarnen zu lassen) – so viel zum »Handlesen« eines Programmierers. Beachten Sie allerdings, dass Sie mit der folgenden Zeile

```
int* iptr1, iptr2;
```

nur einen Zeiger deklariert haben. So erscheint der C-Stil einer Zeiger-Deklaration in puncto Überblick besser geeignet:

```
int *iptr1, *iptr2;
```

Das Sternchen, das bei einem Zeiger verwendet wird, wird als (unärer) *Indirektionsoperator* bezeichnet.

2.1.2 Adresse im Zeiger speichern

Es wurde bisher immer noch nicht erwähnt, wozu ein Zeiger verwendet wird. Ein Zeiger ist im Grunde eine einfache Variable, die eine Speicheradresse im Arbeitsspeicher aufnehmen kann. Das hört sich zunächst nicht spektakulär an, aber im Verlauf des Buches werden Sie Ihre Meinung ändern.

Um die Adresse einer Variablen in einem Zeiger speichern zu können, benötigt man wiederum den unären Adressoperator `&`. Ein Beispiel dazu:

```
int var = 100;
int *iptr;
// iptr zeigt auf var
iptr = &var;
```

Hiermit weisen Sie dem Zeiger `iptr` die Adresse von `var` zu. Die folgende Abbildung soll diesen Vorgang näher demonstrieren.

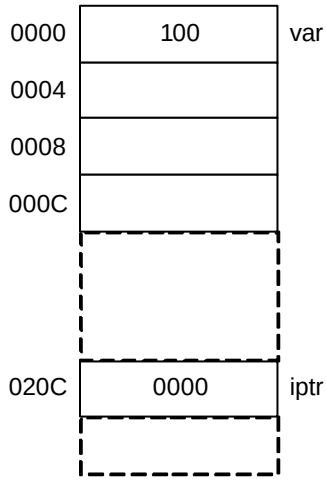

Abbildung 2.2 Adresszuweisung an Zeiger

In dieser Abbildung sehen Sie, dass der Zeiger `iptr` die Adresse der Variablen `var` (hier `0000`) enthält. Man spricht hierbei auch von einer *Indirektion*, weil auf das eigentliche Speicherobjekt (hier `var`) nicht direkt, sondern indirekt über einen Zeiger (`iptr`) zugegriffen wird.

Wollen Sie nun ausgeben, welche Adresse ein Zeiger enthält, so benötigen Sie im Gegensatz zu den normalen Basisdatentypen keinen Adressoperator, da ein Zeiger ja eine Adresse speichert. Verwenden Sie dennoch den Adressoperator, so wird die Speicheradresse des Zeigers selbst ausgegeben:

```
// zeiger2.cpp
#include <iostream>
using namespace std;

int main(void) {
   int var = 100;
   int* iptr;
```

```
   iptr = &var;
   cout << "Adresse von var   : " << &var << '\n';
   cout << "iptr verweist auf : " << iptr << '\n';
   cout << "Adresse von iptr  : " << &iptr << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
Adresse von var   : 0x22ff74
iptr verweist auf : 0x22ff74
Adresse von iptr  : 0x22ff70
```

2.1.3 Zeiger dereferenzieren

Jetzt können Sie zwar Adressen in einem Zeiger von anderen Variablen speichern, aber in der Praxis werden Sie wohl kaum mit Adressen arbeiten wollen, sondern mit echten Werten. Um dies zu realisieren, wird der unäre Indirektionsoperator (*) verwendet. Hierzu das folgende Beispiel:

```
// zeiger3.cpp
#include <iostream>
using namespace std;

int main(void) {
   int var = 100;
   int tmp;
   int* iptr;
   iptr = &var;
   // Dereferenzierung
   tmp = *iptr;
   cout << "Adresse von var   : " << &var << '\n';
   cout << "Wert von var      : " << var << '\n';
   cout << "Adresse von tmp   : " << &tmp << '\n';
   cout << "Wert von tmp      : " << tmp << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
Adresse von var   : 0x22ff74
Wert von var      : 100
Adresse von tmp   : 0x22ff70
Wert von tmp      : 100
```

Nachdem Sie dem Zeiger `iptr` die Adresse von `var` zugeordnet haben, wurde indirekt über

```
tmp = *iptr;
```

auf den Wert von `var` zurückgegriffen. Diese Zeile entspricht somit folgender Anweisung:

```
tmp = var;
```

Die Auflösung der Adresse von der Zeigervariablen wird als Dereferenzierung bezeichnet. Natürlich lässt sich eine solche Dereferenzierung auch anders verwenden:

```
// zeiger4.cpp
#include <iostream>
using namespace std;

int main(void) {
   int var = 100;
   int tmp;
   int* iptr;
   iptr = &var;

   // Dereferenzierung
   tmp = *iptr;
   cout << "Adresse von var   : " << &var << '\n';
   cout << "Wert von var      : " << var << '\n';
   cout << "Adresse von tmp   : " << &tmp << '\n';
   cout << "Wert von tmp      : " << tmp << "\n\n";

   // Dereferenzierung
   *iptr = 200;
   cout << "Adresse von var   : " << &var << '\n';
   cout << "Wert von var      : " << var << '\n';
   cout << "Adresse von tmp   : " << &tmp << '\n';
   cout << "Wert von tmp      : " << tmp << "\n\n";

   // Dereferenzierung
   *iptr -= 5;
   cout << "Adresse von var   : " << &var << '\n';
   cout << "Wert von var      : " << var << '\n';
   cout << "Adresse von tmp   : " << &tmp << '\n';
   cout << "Wert von tmp      : " << tmp << "\n\n";
   return 0;
}
```

Das Programm bei der Ausführung:

```
Adresse von var   : 0x22ff74
Wert von var      : 100
Adresse von tmp   : 0x22ff70
Wert von tmp      : 100

Adresse von var   : 0x22ff74
Wert von var      : 200
Adresse von tmp   : 0x22ff70
Wert von tmp      : 100

Adresse von var   : 0x22ff74
Wert von var      : 195
Adresse von tmp   : 0x22ff70
Wert von tmp      : 100
```

Mit der Anweisung

```
*iptr = 200;
```

wurde im Beispiel der Wert der Variablen `var` indirekt über die Adresse, die aus dem Pointer `iptr` bekannt ist und auf denselben Speicherbereich verweist, verändert. Der Inhalt von `tmp` bleibt unverändert, da diese Variable einer anderen Speicheradresse zugeordnet ist.

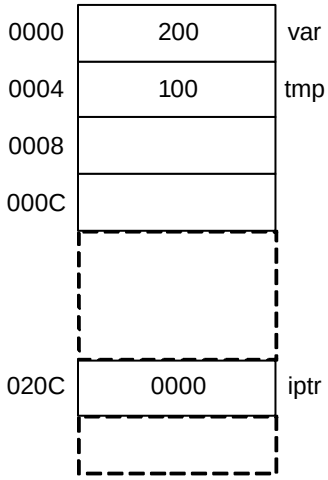

Abbildung 2.3 Auflösung der Adresse einer Zeigervariablen

Die nächste Dereferenzierung mit

```
*iptr -= 5;
```

soll zeigen, dass mit Hilfe des Indirektionsoperators auch arithmetische Operationen möglich sind. Folgende Rechenoperationen können mit einem Zeiger und auf dessen Adresse verwendet werden:

- Ganzzahlwerte erhöhen
- Ganzzahlwerte verringern
- Inkrementieren
- Dekrementieren

Des Weiteren sind bei Verwendung eines Zeigers natürlich auch die Vergleichsoperatoren <, >, !=, ==, <= und >= erlaubt. Die Verwendung ist aber nur sinnvoll, wenn die Zeiger auf Elemente eines Arrays (siehe Abschnitt 2.3, »Arrays«) zeigen, oder bei Parametern von Funktionen, um Arbeitsspeicher zu sparen.

Ein Problem im Umgang mit Zeigern entsteht, wenn Sie einen Zeiger dereferenzieren, der auf kein gültiges Objekt zugreift – also keinen speziellen Wert erhalten hat. Ein Beispiel dazu:

```
int* iptr;

*iptr = 100;
```

Hier wurde dem Zeiger indirekt der Wert `100` zugewiesen. Da ihm zuvor aber noch kein Speicherobjekt übergeben wurde, auf das dieser verweist, wird hierbei auf eine Zufallsadresse (meistens ein zufälliges Bit-Muster, das vom Linker erzeugt wurde) zugegriffen. Befindet sich diese Speicheradresse außerhalb des Speicherbereichs des Programms, so haben Sie eine Speicherschutzverletzung.

Hierfür wird der Zeiger gewöhnlich mit `0` initialisiert. Damit wird angezeigt, dass dieser Zeiger noch auf kein gültiges Speicherobjekt zeigt, und somit darf er auch nicht dereferenziert werden:

```
// zeiger5.cpp
#include <iostream>
using namespace std;

int main(void) {
   int* iptr = 0;
   if ( iptr == 0 ) {
      // iptr verweist noch auf kein Objekt
   }
   return 0;
}
```

C-Programmierer werden sich hier fragen, wo die Konstante `NULL` geblieben ist. Da nicht genau gesagt werden kann, ob `NULL` mit `0` oder `(void*)0` definiert ist, kann Letzteres bei C++ zu Problemen führen, weil man ein Typcasting vornehmen muss, um den `void*`-Wert einem Zeiger zuzuweisen.

Überblick

Um jetzt ein wenig das Tempo herauszunehmen, hier eine kurze Zusammenfassung: Jede Variable besitzt eine Adresse im Speicher. Um auf die Adresse einer Variablen zuzugreifen, wird der Adressoperator `&` verwendet. Diese Adresse kann man auch in einem Zeiger speichern. Ein Zeiger besteht aus dem Typ des Speicherobjekts gefolgt vom Indirektionsoperator `*` und dem Zeiger-Namen. Um auf den Wert zuzugreifen, der in der im Zeiger gespeicherten Adresse abgelegt ist, muss der Indirektionsoperator `*` verwendet werden. Man spricht hierbei von einer Dereferenzierung.

2.1.4 Zeiger, die auf andere Zeiger verweisen

Natürlich können Zeiger auch auf andere Zeiger verweisen:

```
int *iptr1;
int *iptr2;
...
iptr1 = iptr2;
```

Hiermit verweisen beide Zeiger auf dieselbe Adresse. Ein einfaches Beispiel dazu:

```
// zeiger6.cpp
#include <iostream>
using namespace std;

int main(void) {
   int* iptr1 = 0;
   int* iptr2 = 0;
   int wert = 100;
   iptr1 = &wert;
   iptr2 = iptr1;
   cout << "iptr1 verweist auf " << iptr1 << '\n';
   cout << "iptr2 verweist auf " << iptr2 << '\n';
   cout << "Adresse von wert : " << &wert << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
iptr1 verweist auf 0x22ff6c
iptr2 verweist auf 0x22ff6c
Adresse von wert : 0x22ff6c
```

Beide Zeiger verweisen also auf die Adresse von `wert`. Das bedeutet in der Praxis: Wenn einer dieser Zeiger den Wert mit dem Indirektionsoperator `*` manipuliert, so bezieht sich die Veränderung immer indirekt auf die Variable `wert`:

```
// zeiger7.cpp
#include <iostream>
using namespace std;

int main(void) {
   int* iptr1 = 0;
   int* iptr2 = 0;
   int wert = 100;
   iptr1 = &wert;
   iptr2 = iptr1;

   cout << "wert : " << wert << '\n';
   // indirekter Zugriff über *
   cout << "wert : " << *iptr1 << '\n';
   cout << "wert : " << *iptr2 << "\n\n";

   // Dereferenzierung
   *iptr1 = 200;
   cout << "wert : " << wert << '\n';
   // indirekter Zugriff über *
   cout << "wert : " << *iptr1 << '\n';
   cout << "wert : " << *iptr2 << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
wert : 100
wert : 100
wert : 100

wert : 200
wert : 200
wert : 200
```

Bestimmt ist Ihnen aufgefallen, dass bei der Übergabe von der Adresse eines Zeigers zu einem anderen kein Adressoperator verwendet wurde. Dies ist bei den Zeigern nicht nötig, da der `Wert` eines Zeigers schon eine Adresse im Speicher darstellt und ein Zeiger auch einen `Wert` als Adresse erwartet.

2.1.5 Dynamisch Speicherobjekte anlegen und zerstören – »new« und »delete«

Bisher haben Sie Variablen verwendet, die beim Ausführen Ihres Anweisungsblocks automatisch angelegt und beim Verlassen wieder gelöscht wurden. Der Vorteil hierbei ist natürlich, dass man sich um nichts kümmern muss. Was ist aber, wenn Sie nicht genau sagen können, wie viele Objekte Sie für das Programm benötigen? Zwar gibt es hierbei Dinge wie Arrays, aber diese sind ebenfalls wieder auf eine bestimmte Größe beschränkt. Man kann natürlich ein Array mit einer enormen Größe belegen, aber man sollte bedenken, dass das Programm dann auch diese Größe an Speicherplatz benötigt.

Hier haben Sie auch gleich einen Hauptnutzen der Zeiger. Da Sie bei den Zeigern mit Adressen arbeiten, können Sie Speicherobjekte zur Laufzeit des Programms dynamisch anlegen und auch wieder zerstören. Allerdings sollten Sie an dieser Stelle gewarnt sein, denn wenn Sie die Verwaltung von Speicherobjekten selbst übernehmen, müssen Sie auch entsprechend Platz dafür im Speicher reservieren und diesen auch wieder freigeben (zerstören), wenn er nicht mehr benötigt wird.

Allerdings ist Speicher nicht gleich Speicher. Generell unterscheidet man bei einem laufenden Programm folgende Speicherbereiche:

- Codespeicher – dieser wird in den Arbeitsspeicher geladen, und von dort aus werden die Maschinenbefehle der Reihe nach in den Prozessor (genauer in die Prozessor-Register) geschoben und ausgeführt.
- Datenspeicher – darin befinden sich alle statischen Daten, die bis zum Programmende verfügbar sind (globale und statische Variablen).
- Stack-Speicher – im Stack werden die Funktionsaufrufe mit ihren lokalen Variablen verwaltet. Auf den Stack wurde ja bereits eingegangen.
- Heap-Speicher – dem Heap-Speicher steht der verbleibende Speicherplatz zur Verfügung, und diesem gebührt auch das Hauptinteresse in diesem Abschnitt. Mit ihm funktioniert auch die dynamische Speicheranforderung. Der Heap funktioniert ähnlich wie der Stack. Bei einer Speicheranforderung erhöht sich der Heap-Speicher, und bei einer Freigabe wird er wieder verringert. Wenn Speicher angefordert wurde, wird die Anfangsadresse des Speicherblocks zurückgegeben, wenn genug Speicher vorhanden war. Und wenn von Adressen die Rede ist, dann ist auch ein Zeiger nicht weit entfernt.

  Der Heap hat außerdem den Vorteil, dass ein einmal reservierter Speicherplatz verfügbar bleibt, bis dieser wieder freigegeben wird. Der Vorteil kann allerdings auch zu einem erheblichen Nachteil werden. Reservieren Sie zum Beispiel Speicher vom Heap in einer Funktion und beenden diese Funktion, sollten Sie immer die Adresse des reservierten Speichers in einem Zeiger speichern, da,

anders als beim Stack, dieser reservierte Speicher nach dem Ende der Funktion erhalten bleibt. Haben Sie keinen Zeiger, der auf diesen Speicherbereich verweist, dann haben Sie ein schönes Speicherloch (engl.: *Memory Leak*) in den Heap-Speicher »geschossen«, der nicht mehr verwendet werden kann. Eine andere Lösung des Problems ist natürlich, diesen Speicher wieder freizugeben.

[»] **Hinweis**

Die Beschreibung des Speicherkonzepts wurde hier erheblich vereinfacht dargestellt. Letztendlich ist es für den Programmierer nur wichtig zu wissen, dass das dynamische Speichermanagement im Heap stattfindet.

new – dynamische Objekte anlegen

Um Speicherobjekte dynamisch zur Laufzeit des Programms anzulegen, wird der Operator `new` verwendet. Hinter `new` wird der Typ des anzulegenden Objekts angegeben. Das System fordert daraufhin so viel Speicherplatz an, wie der angegebene Typ benötigt. Bei Erfolg erhalten Sie als Rückgabewert eine Adresse des erfolgreich reservierten Speicherplatzes. Diesem Rückgabewert weisen Sie natürlich einen Zeiger zu. Die Syntax lautet:

```
Typ* zeiger = new Typ;
```

Oder in der Praxis:

```
int *iptr;
iptr = new int;
```

Bei alten C++-Compilern liefert `new` im Falle eines Fehlers (wenn kein Speicher für das Objekt reserviert werden konnte) `0` zurück. Also sollten Sie dabei auch überprüfen, ob der Zeiger nicht `0` ist.

```
int *iptr;
iptr = new int;

if ( iptr == 0 ) {
   // Fehler bei der Speicherreservierung
}
else {
   // Speicherreservierung erfolgreich
}
```

Neuere C++-Compiler arbeiten hierbei auch schon mit Ausnahmen (Exceptions; siehe Kapitel 6, »Exception-Handling«). Dabei gibt es drei verschiedene Varianten des Operators `new` (bzw. auch `new []`):

```
void* operator new( size_t )   throw(bad_alloc);
void* operator new( size_t, const nothrow_t&)   throw();
void* operator new( size_t, void* )   throw();
```

Lassen Sie sich jetzt nicht von der Syntax dieser drei `new`-Versionen abschrecken. Die erste Variante ist Standard und wird immer verwendet, wenn Sie keine andere Variante angeben. Bei dieser Variante wird die Ausführung des Programms beendet, wenn kein Speicherplatz im Heap angefordert werden konnte. Die zweite Variante gibt im Falle eines Fehlers `0` als Rückgabewert zurück, das entspricht dem Verhalten älterer C++-Compiler. Mit der dritten Variante haben Sie einen weiteren Parameter, und dieser wird verwendet, wenn Sie persistente Speicherobjekte anlegen wollen (Placement-`new`-Operator). Darauf wird allerdings in diesem Buch nicht eingegangen, weil es in der Praxis recht selten verwendet wird.

Welche Variante Sie verwenden wollen, bleibt natürlich Ihnen überlassen. Meistens wird die erste Variante eingesetzt, in der bei fehlendem Heap-Speicher das Programm beendet wird:

```
// Speicher für eine int-Zahl reservieren
int* iptr = new int;
// Speicher für eine double-Zahl reservieren
double* dptr = new double;
```

Wollen Sie die zweite Variante verwenden, bei der die Ausführung des Programms nicht beendet wird, um es zum Beispiel nochmals zu versuchen oder sonstige Aufräumarbeiten durchzuführen, so wird dies nach neuem C++-Standard wie folgt realisiert:

```
int* iptr = new(nothrow) int;

if( iptr ) {
   // Speicherreservierung erfolgreich
}
else {
   // Fehler bei der Speicherreservierung
}
```

Mit der Verwendung der vordefinierten Konstante `nothrow` geben Sie an, dass Sie die Ausnahmebehandlung des `new`-Operators nicht verwenden wollen.

Natürlich sollte Ihnen auch klar sein, dass Sie auf ein mit `new` reserviertes Speicherobjekt nur über den Zeiger selbst (also mit dem Indirektionsoperator) zugreifen bzw. einen Wert ablegen können:

```
// Speicher für eine int-Zahl reservieren
int* iptr = new int;

// Wert 100 am Zeiger iptr ablegen
*iptr = 100;
```

Allerdings sollte man niemals die Adresse des Speicherobjekts »verlieren« – was in einem einfachen Beispiel eigentlich kaum möglich ist, bei umfangreichen Datenstrukturen aber recht schnell passieren kann.

[»] **Hinweis**

Bedenken Sie immer: Wenn Sie die Adresse verlieren, haben Sie keine Möglichkeit mehr, das Speicherobjekt zu verwenden, obwohl es im Heap-Speicher weiterhin vorhanden ist.

delete – dynamische Objekte zerstören

Wie bereits mehrmals erwähnt, werden dynamisch erzeugte Objekte nicht mehr von selbst freigegeben und müssen bei Nichtgebrauch explizit wieder freigegeben (oder auch zerstört) werden. Hierzu wird der Operator `delete` (bzw. `delete []`) verwendet.

```
delete zeiger;
```

Der Operator `delete` gibt immer das Speicherobjekt auf der rechten Seite, das ein Zeiger sein muss, wieder frei. Beachten Sie aber, dass nicht festgeschrieben ist, wie `delete` das Objekt freigibt. Es muss also nicht sein, dass der Wert der Adresse, der freigegeben wird, auf `0` gesetzt wird. `delete` »markiert« den Speicher im Heap lediglich als frei für die erneute Verwendung.

Zerstören Sie mit `delete` einen Zeiger, der auf keinen gültigen Speicherplatz zeigt, so ist das Verhalten undefiniert (also nicht vorhersehbar).

[»] **Hinweis**

Andere Programmiersprachen, wie zum Beispiel Java, verfügen über eine automatische Freispeicherverwaltung, die *Garbage Collection*, bei der Sie sich nicht mehr um das Freigeben von angefordertem Speicher kümmern müssen – natürlich geht so etwas auf Kosten der Performance.

Hierzu ein einfaches Beispiel, das die Verwendung der Operatoren `new` und `delete` demonstriert:

```
// zeiger8.cpp
#include <iostream>
using namespace std;
```

```
int main(void) {
   int* iptr1 = new int;
   int* iptr2 = new int;
   int tmp;
   *iptr1 = 100;
   *iptr2 = 200;
   cout << "Wert von *iptr1 : " << *iptr1 << '\n';
   cout << "Wert von *iptr2 : " << *iptr2 << "\n\n";

   // Werte tauschen
   tmp = *iptr2;
   *iptr2 = *iptr1;
   *iptr1 = tmp;
   cout << "Wert von *iptr1 : " << *iptr1 << '\n';
   cout << "Wert von *iptr2 : " << *iptr2 << '\n';

   // Speicher wieder freigeben
   delete iptr1;
   delete iptr2;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Wert von *iptr1 : 100
Wert von *iptr2 : 200
Wert von *iptr1 : 200
Wert von *iptr2 : 100
```

Die meisten modernen Betriebssysteme geben reservierten, aber nicht freigegebenen Speicher beim Beenden eines Programms wieder frei, obwohl dies nicht vom ANSI/ISO-Standard gefordert wird.

Sicherlich stellen Sie sich auch die Frage, was passiert, wenn Sie für andere Objekte als Basisdatentypen Speicher reservieren. Woher weiß `delete`, wie groß der Speicherblock im Heap ist, der zerstört werden muss? Darum müssen Sie sich nicht kümmern, das ist wiederum die Aufgabe der Speicherverwaltung des Betriebssystems.

Hinweis **[«]**

Die Funktionen `malloc()`, `realloc()`, `calloc()` bzw. `free()` (zum Freigeben von Speicher) in einem Quellcode sind die Standard-C-Funktionen zur dynamischen Speicherverwaltung, die in der Header-Datei <*cstdlib*> (in C bekannt als <*stdlib.h*>) definiert sind. Da C eine Untermenge von C++ ist und einige Programme auch diese Funktionen noch verwenden, sei für weitere Recherchen auf mein Buch »C von A bis Z« verwiesen, das Sie von Galileo Press beziehen können und auch als Openbook auf der Buch-CD finden (oder auf *http://www.pronix.de/*).

2.1.6 void-Zeiger

[»] **Hinweis**

Wer keine Kenntnisse in C besitzt, kann diesen Abschnitt überspringen, da er sich vorwiegend an C-Programmierer richtet.

`void`-Zeiger (`void *`) wurden in C gerne verwendet, um Funktionen zu implementieren, die verschiedene Arten von Datentypen verarbeiten können. `void` allein hat relativ wenig »Nutzen« und zeigt nur an, dass kein spezieller Typ verwendet wird. `void`-Zeiger hingegen können wie normale Zeiger eine gültige Adresse speichern und sind von keinem Typ abhängig.

In C konnte ein `void`-Zeiger ohne Schwierigkeit dereferenziert werden – C++ hingegen schreibt eine Typumwandlung vor, wenn man den Zeiger dereferenzieren will. Zwar wird auf eine solche Typumwandlung noch extra eingegangen (siehe Abschnitt 3.7, »Typumwandlung«), aber hier soll schon mal ein solcher Vorgang in der Praxis gezeigt werden:

```
// zeiger9.cpp
#include <iostream>
using namespace std;

int main(void) {
   void *vptr;
   int ivar = 100;
   vptr = &ivar;
   // Umständliche Dereferenzierung:
   cout << *(static_cast<int*>(vptr)) << '\n';
   return 0;
}
```

Da C++ mit den Templates bessere Alternativen zu den `void`-Zeigern bietet, um generische Datenstrukturen zu erzeugen, und weil die Typprüfung bei der Verwendung von `void`-Zeigern durch den Compiler fast unmöglich ist, sei von `void`-Zeigern in der C++-Programmierung abgeraten (wenn möglich).

2.1.7 Konstante Zeiger

Auch bei den Zeigern kann mit dem Schlüsselwort `const` ein Read-only-Zeiger verwendet werden. Aber ab wann haben wir einen konstanten Zeiger? Das ist auf den ersten Blick nicht so leicht zu durchschauen. Ein einfaches Beispiel hierfür:

```
const char *cptr = "ABC";
cptr =   "XYZ";   // Ok, kein Fehler!!!
*cptr = 'X';      // Fehler *cptr ist eine Konstante
```

Warum kann `cptr` verändert werden und `*cptr` nicht? Ganz einfach: `cptr` ist nur ein Zeiger, der auf ein `const char`-Array zeigt. Die Daten, auf die `*cptr` zeigt, können dagegen nicht verändert werden, weil diese konstant sind. Verändern wir nun die Position des Indirektionsoperators:

```
const *char cptr = "ABC";
cptr =   "XYZ";   // Fehler cptr ist konstant
*cptr = 'X';      // Ok, *cptr ist ein char
```

Jetzt erhalten Sie das umgekehrte Bild. Nun ist der Zeiger konstant, und die Daten, auf die er zeigt (die Zeichen), können geändert werden. Somit sollten Sie immer die Position des Indirektionsoperators beachten.

Halten Sie hierbei immer die Begriffe »Daten« und »Zeiger« auseinander – dies sind zwei verschiedene Dinge!

Wollen Sie letztendlich, dass sowohl der Zeiger als auch die Daten konstant und somit nicht mehr veränderbar sind, dann müssen Sie nur Folgendes schreiben:

```
const char *const cptr = "ABC";
cptr =   "XYZ";   // Fehler cptr ist konstant
*cptr = 'X';      // Fehler *cptr ist konstant
```

Hinweis [«]

Bevor sich jetzt einige C-Fanatiker aufmachen, um Protestmails zu verfassen, dass man den Schreibschutz sehr wohl umgehen kann, soll hier noch darauf hingewiesen werden, dass man theoretisch den Zeiger immer irgendwie »verbiegen« kann, um den Schreibschutz zu unterlaufen. Aber darauf wird hier nicht eingegangen.

2.2 Referenzen

Eine Referenz in C++ ist nichts anderes als ein anderer Name (Alias-Name) für ein Speicherobjekt. Beim Anlegen einer Referenz muss diese mit einem bereits bestehenden Objekt initialisiert werden. Die Deklaration von Referenzen erfolgt, indem hinter dem Typ und vor dem Namen das Begrenzungszeichen `&` steht. Auch hierbei kann, wie schon bei den Zeigern, das Zeichen `&` unmittelbar hinter dem Typ oder vor dem Namen stehen.

```
int var = 100;
int  &rvar1 = var;
int&  rvar2 = var;
```

Allerdings sollte man bei der zweiten Verwendung nicht den Fehler machen zu glauben, dass durch die Angabe `Typ&` die weiteren Objekte ebenfalls Referenzen

sind. In diesem Beispiel ist nur `rvar1` eine Referenz. `rvar2` ist eine normale `int`-Variable:

```
int var = 100;
int& rvar1=var, rvar2=var;
```

Eine gewisse Ähnlichkeit von Referenzen und Zeigern kann man nicht leugnen. Kein Wunder, Referenzen werden ja intern mit Zeigern realisiert. Dennoch sollte man sich immer vor Augen halten, dass Referenzen nur Alias-Namen auf andere Speicherobjekte sind. Zeiger hingegen können Adressen von anderen Speicherobjekten aufnehmen.

Alias

Ein Alias (lateinisch für »ein anderer«) bezeichnet einen Ersatznamen (Pseudonym).

Hierzu ein einfaches Beispiel, das die Referenzen demonstrieren soll:

```
// ref1.cpp
#include <iostream>
using namespace std;

int main(void) {
   int var;
   int& rvar = var;

   var = 100;
   cout << "var  : " << var << '\n';
   cout << "rvar : " << rvar << '\n';

   rvar = 200;
   cout << "var  : " << var << '\n';
   cout << "rvar : " << rvar << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
var  : 100
rvar : 100
var  : 200
rvar : 200
```

An der Zeile

```
rvar = 200;
```

kann man eindeutig den Unterschied zwischen Referenzen und Zeigern erkennen. Bei einem Zeiger hätten Sie hier nur mit dem Indirektionsoperator auf die Variable `var` indirekt zugreifen können. Bei Referenzen genügt der Name, da dieser nichts anderes als ein Synonym für die eigentliche Variable ist. Am besten lässt sich dieser Sachverhalt demonstrieren, wenn man den Adressoperator verwendet.

```
// ref2.cpp
#include <iostream>
using namespace std;

int main(void) {
   int var;
   int& rvar = var;
   cout << "&var  : " << &var << '\n';
   cout << "&rvar : " << &rvar << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
&var  : 0x22ff74
&rvar : 0x22ff74
```

Da Referenzen bei ihrer Erzeugung initialisiert werden, arbeiten diese immer als Synonyme für das eigentliche Ziel. Daher gibt es auch keine Möglichkeit, die Adresse einer Referenz zu ermitteln.

Referenzen können aber nur einmal zugewiesen werden. Der Versuch einer erneuten Zuweisung hätte nur den Effekt, dass Sie den Wert der Zielreferenz, mit dem Sie die Referenz am Anfang initialisiert haben, verändern.

```
int var1 = 100;
int var2 = 200;
int& rvar = var1;
// neue Zuweisung?
rvar = var2;
```

Der Versuch einer zweiten Zuweisung hat hier lediglich den Effekt, dass der Wert von `var1` auf `200` verändert wird.

Das Anwendungsgebiet von Referenzen liegt vorwiegend in der Parameterübergabe und der Wertrückgabe von Funktionen. Im Beispiel des Rückgabewerts muss so kein Speicherobjekt dafür angelegt werden, und es wird nur ein Alias für ein bereits bestehendes Objekt zurückgegeben. Mehr dazu erfahren Sie in den Abschnitten 2.6.3, »Call by reference mit Referenzen nachbilden«, und 2.7.2, »Referenz als Rückgabewert«.

2.3 Arrays

Mit den Arrays können Sie eine geordnete Folge von Werten eines bestimmten Typs abspeichern und bearbeiten. In vielen Büchern werden Arrays auch als Felder oder Vektoren bezeichnet.

Die gleichartigen Objekte werden sequentiell im Speicher abgelegt – darauf können Sie sich immer verlassen. Das bedeutet natürlich, dass ein Array mit fünf `long`-Werten auf einem 32-Bit-Rechner 20 Bytes – 5 x 4 Bytes (für `long`) = 20 Bytes – belegt.

[»] **Hinweis**

Auch wenn sich die Beispiele hier zunächst nur auf die Basisdatentypen beziehen, soll nicht der Eindruck entstehen, dass sich Arrays nur auf diese anwenden lassen. Aber hierauf wird im Verlauf des Buches noch explizit eingegangen.

2.3.1 Arrays deklarieren

Hier folgt nun die Syntax zur Deklaration eines Arrays:

```
Typ ArrayName[n];
```

Mit `Typ` geben Sie an, von welchem Typ die Elemente des Arrays sind. Der `ArrayName` ist frei wählbar nach denselben Vorschriften wie ein Bezeichner für Variablen. Mit `n` geben Sie die Anzahl der Elemente an, die dieses Array vom `Typ` aufnehmen kann, also den Indexwert. Arrays mit unterschiedlichen Typen gibt es nicht in C++.

Die einzelnen Elemente werden mit dem Array-Namen und dem entsprechenden Indexwert in eckigen Klammern angesprochen. Der Indexwert muss eine Ganzzahl sein und fängt immer bei 0 an zu zählen! Hierzu einige Beispiele von Arrays:

```
int iarray[10];        // 10  int-Werte
float farray[50];      // 50  float-Werte
double darray[100];    // 100 double-Werte
```

Noch ein einfaches Beispiel:

```
int var[5];
```

Im Speicher ergibt sich somit folgendes Bild:

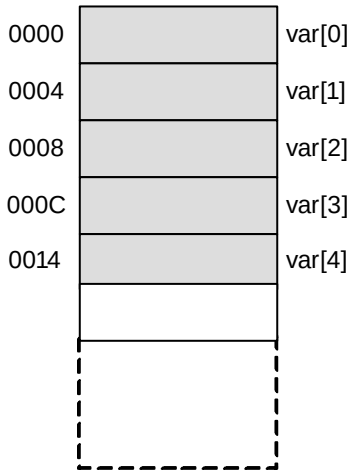

Abbildung 2.4 Array mit fünf Elementen

Anhand des Typs erkennt der Compiler von selbst, wie viel Speicher für das Array benötigt wird.

2.3.2 Arrays initialisieren

Die einzelnen Werte eines Arrays können Sie nun mit dem Array-Namen und dem entsprechenden Indexwert zuweisen.

```
int var[5];

var[0] = 2;
var[1] = 4;
var[2] = 6;
var[3] = 8;
var[4] = 10;
```

Oder auch in einer Schleife:

```
// array1.cpp
#include <iostream>
using namespace std;

int main(void) {
   int var[5];
   for( int i=0; i < 5; i++ ) {
      var[i] = (i+1) * 2;
```

```
    }
    for( int i=0; i < 5; i++ ) {
       cout << "var[" << i <<"]:" << var[i] << "\n";
    }
    return 0;
}
```

In diesem Beispiel wurde an alle fünf Elemente ein Wert mit Hilfe des Indizierungsoperators (`[]`) übergeben und anschließend auch wieder ausgegeben:

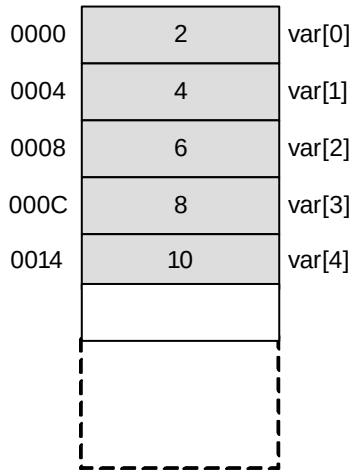

Abbildung 2.5 Ein initialisiertes Array mit fünf Elementen

Arrays lassen sich auch, anders als eben gezeigt, direkt bei der Deklaration, initialisieren. Die Werte müssen dabei wie folgt zwischen geschweiften Klammern stehen:

```
int var[5] = { 2, 4, 6, 8, 10 };
```

Bei einem so initialisierten Array können Sie die Indexgröße des Arrays auch weglassen:

```
int var[] = { 2, 4, 6, 8, 10 };
```

Der Compiler kümmert sich dann darum, dass genügend Speicherplatz zur Verfügung steht. Die einzelnen Werte werden immer mit einem Komma getrennt und stehen in geschweiften Klammern. Schreibt man hingegen

```
int var[5] = { 2, 4 };
```

so ist dies kein Fehler. Hiermit werden praktisch die ersten beiden Array-Elemente mit den Werten `2` und `4` initialisiert – alle anderen Elemente erhalten auto-

matisch den Wert 0. Sofern Sie also bei einem Array alle Werte mit 0 vorbelegen wollen, müssen Sie hierfür nicht extra eine Schleife verwenden.

```
// Unnötig
for( int i=0; i < 5; i++ ) {
      var[i] = 0
}
```

Es genügt dagegen folgende Initialisierung:

```
int var[5] = { 0 };
```

2.3.3 Bereichsüberschreitung von Arrays

Hier kommen wir nun zu einem Punkt, der einem in der Vergangenheit schon viele Fehler beschert hat. Anhand der Indexnummer können Sie erkennen, dass hier bei 0 mit dem »zählen« begonnen wurde und mit 4 das letzte Element beziffert wird. Genau betrachtet sind es ja fünf Zahlen (0, 1, 2, 3, 4) – aber meistens entspricht dies nicht der logischen Denkweise (erster Schultag, erster Platz, erster Sieger, erster Tag usw.). Aber gerade in Schleifen wird hierbei gerne Folgendes geschrieben:

```
// !!! Bereichsüberschreitung !!!
for( int i=0; i <= 5; i++ ) {
   var[i] = (i+1) * 2;
}
```

Hier wurde statt auf »kleiner« 5 auf »kleiner-gleich« 5 geprüft. Viele Compiler machen dabei anstandslos mit, und im Speicher würde sich folgender Zustand finden:

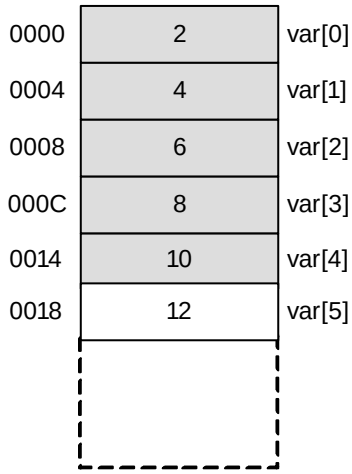

Abbildung 2.6 Bereichsüberschreitung eines Arrays

Das weitere Verhalten des Programms ist in diesem Fall undefiniert. Sobald zum Beispiel eine andere Variable diesen Speicherbereich (hier `0018`) bekommt und verwendet, wird dieser Wert gnadenlos überschrieben. Besonders fatal wirkt sich das bei `char`-Arrays aus, da hier oft die Nullterminierung überschrieben wird. Somit liegt es praktisch in Ihrer Verantwortung, den Bereich eines Arrays nicht zu überschreiten.

Auf manchen Systemen gibt es eine Compiler-Option (*range checking*), mit der ein solcher Über- bzw. Unterlauf eines Arrays zur Laufzeit des Programms geprüft wird. Das fertige Programm sollte allerdings nicht mehr mit dieser Option übersetzt werden, da dies zu einem schlechten Laufzeitverhalten führt.

2.3.4 Anzahl der Elemente eines Arrays ermitteln

Die Anzahl der Elemente eines Arrays können Sie mit dem `sizeof`-Operator ermitteln. Dieser liefert Ihnen die Größe eines Typs in Bytes zurück. Auf das Array allein angewandt, liefert der `sizeof`-Operator allerdings nur die gesamte Größe des Arrays zurück. Teilt man diesen Wert durch die Größe eines einzelnen Typs, dann erhält man die Anzahl der Elemente zurück.

```
// array2.cpp
#include <iostream>
using namespace std;

int main(void) {
   int var[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
   cout << "Gesamtgröße : " << sizeof(var)
        << " Bytes\n";
   cout << "Einzelgröße : " << sizeof(var[0])
        << " Bytes\n";
   cout << "Anzahl Elemente : "
        << sizeof(var)/sizeof(var[0]) << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
Gesamtgröße : 32 Bytes
Einzelgröße : 4 Bytes
Anzahl Elemente : 8
```

2.3.5 Array-Wert von Tastatur einlesen

Das Einlesen von Array-Werten von der Tastatur funktioniert genauso wie bei normalen Variablen. Es muss lediglich das Indexfeld (mit Hilfe des Indizierungsoperators) verwendet werden, für das Sie den Wert vorgesehen haben.

```
// array3.cpp
#include <iostream>
using namespace std;

int main(void) {
   int var[5];
   for( int i=0; i < 5; i++ ) {
      cout << "Wert eingeben : ";
      cin >> var[i];
   }
   cout << "Die eingegebenen Werte waren :\n";
   for( int i=0; i < 5; i++ ) {
      cout << var[i] << '\n';
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Wert eingeben : 55
Wert eingeben : 43
Wert eingeben : 45
Wert eingeben : 11
Wert eingeben : 15
Die eingegebenen Werte waren :
55
43
45
11
15
```

2.3.6 Mehrdimensionale Arrays

Selbstverständlich kann man in C++ auch ein Array mit mehr als nur einem Index verwenden, also ein mehrdimensionales Array. Bei einem zweidimensionalen Array hat man dann zwei Indizierungsoperatoren statt – wie bisher – einen.

```
int spielfeld[2][4];
```

Hier wurde ein zweidimensionales Array mit dem Namen `spielfeld` deklariert, das 2×4 Felder besitzt. Man kann sich das Ganze wie bei einer Tabellenkalkulation vorstellen. Der erste Index dient als »Zeile« und der zweite als »Spalte«. Das Gleiche hätten Sie übrigens auch mit

```
int spielfeld[8];
```

realisieren können – nur ist es hierbei nicht so offensichtlich, worauf das Ganze hinauslaufen soll.

Die Anzahl der Dimensionen ist hier allerdings nicht auf zwei beschränkt, doch machen weitere Dimensionen, zum Beispiel eine dritte, in der Praxis kaum noch Sinn.

Die Initialisierung von mehrdimensionalen Arrays erfolgt im Prinzip genauso wie bei einem eindimensionalen Array über die Indizierungsoperatoren. Wollen Sie zum Beispiel auf dem `spielfeld` in der ersten Zeile und dritten Spalte einen Wert zuweisen, können Sie dies wie folgt machen (beachten Sie bitte auch hier, dass bei 0 angefangen wird zu zählen):

```
// 1.Zeile, 3.Spalte
spielfeld[0][2] = 100;
```

Natürlich kann man auch ein mehrdimensionales Array beim Anlegen initialisieren:

```
int spielfeld[2][4] = {  { 11, 22, 33 ,44 },
                         { 55, 66, 77, 88 } };
```

Tatsächlich werden aber die inneren geschweiften Klammern vom Compiler ignoriert, sie dienen lediglich der Übersichtlichkeit. Der Compiler macht aus dieser Initialisierung Folgendes:

```
int spielfeld[2][4] = { 11, 22, 33 ,44 ,55, 66, 77, 88 };
```

Auch für die mehrdimensionalen Arrays gilt, dass alle nicht initialisierten Felder mit dem Wert `0` vorbelegt werden – vorausgesetzt, es wurde mindestens ein Feld mit einem Wert belegt:

```
int spielfeld[2][4] = {  { 11 },
                         { 55, 66} };
```

Hier wurden nur die Felder `[0][0]`, `[1][0]` und `[1][1]` mit Werten belegt. Alle anderen Felder haben den Wert `0`.

Das folgende Beispiel demonstriert Ihnen, wie Sie von der Tastatur aus Werte an ein zweidimensionales Array zuweisen können.

```
// array4.cpp
#include <iostream>
using namespace std;

int main(void) {
   int spielfeld[2][4];
   for (int i=0; i < 2; i++) {
      for (int j=0; j < 4; j++) {
         cout << "Eingabe: ";
         cin >> spielfeld[i][j];
      }
   }
   cout << "Folgende Werte wurden eingegeben :\n";
   for (int i=0; i < 2; i++) {
      for (int j=0; j < 4; j++) {
         cout << spielfeld[i][j] << " ";
      }
      cout << '\n';
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Eingabe: 11
Eingabe: 22
Eingabe: 33
Eingabe: 44
Eingabe: 55
Eingabe: 66
Eingabe: 77
Eingabe: 88
Folgende Werte wurden eingegeben :
11 22 33 44
55 66 77 88
```

In der Praxis werden mehrdimensionale Arrays bei verschiedenen Arten von Berechnungen benötigt oder auch bei 2D-Darstellungen von Grafiken.

2.4 Zeichenketten (C-Strings) – char-Array

C-Programmierer sind mit den klassischen C-Strings bestens vertraut. Allerdings verwendet man bei modernen C++-Programmen keine C-Strings mehr, sondern die C++-Klasse `string`. Wozu also »alte Geister« wecken, werden Sie sich fragen.

Der Hauptgrund ist, dass viele C-Programme mittlerweile in C++-Programme umgeschrieben wurden, und dabei wurde die Verwendung von C-Strings häufig beibehalten. Sofern die Ausführgeschwindigkeit eine Rolle spielen sollte, sind C-Strings erheblich effizienter, da hier der Verwaltungsaufwand geringer ist.

[»] **Hinweis**

Auf die Verwendung der C++-Klasse `string` wird in Abschnitt 7.1, »Die String-Bibliothek (string-Klasse)«, noch ausführlich eingegangen.

2.4.1 C-String deklarieren und initialisieren

Ein C-String ist auch ein Array, nur dass es sich hier um eine Kette von einzelnen `char`-Zeichen mit einer abschließenden 0 (String-Terminierungszeichen `\0` oder `0x00`) handelt. Daher ist die Deklaration eines `char`-Arrays identisch mit der bisher bekannten Form der Array-Deklaration:

```
char carray[100];
```

Damit wird ein Array angelegt, das 100 einzelne Zeichen speichern kann:

```
const char carray[100] = { 'H', 'a', 'l', 'l', 'o', ' ',
                           'W', 'e', 'l', 't', '\n', '\0' };
```

In solch einem Fall ist es nicht nötig, die Größe des Arrays mit anzugeben:

```
const char carray[] = { 'H', 'a', 'l', 'l', 'o', ' ',
                        'W', 'e', 'l', 't', '\n', '\0' };
```

Allerdings ist diese Schreibweise mit den einzelnen Zeichen recht umständlich. Daher erlaubt C/C++ auch die folgende gängige Kurzform:

```
const char carray[] = "Hallo Welt\n";
```

Hier hat man auch gleich den Vorteil, dass das String-Terminierungszeichen (`\0`) nicht hinzugefügt werden muss. Bei der Verwendung von doppelten Hochkommata macht der Compiler dies automatisch.

Somit besitzt das `char`-Array `carray` Platz für zwölf Elemente. Hier liegt jedoch eine mögliche Fehlerquelle und somit auch wieder eine der Schwächen der C-Strings. Da man elf Zeichen sieht, vergisst man gerne das zwölfte, das String-Terminierungszeichen, das das Ende des Strings anzeigt. Wenn Sie also Speicherplatz für einen C-String reservieren müssen, bedenken Sie immer, dass Sie Speicher für n+1 Zeichen reservieren!

2.4.2 C-String einlesen

Es wurde bereits erwähnt, dass man nicht ohne weiteres mit `cin` einen String einlesen kann. Man kann schon, aber es besteht zum einen das Problem, dass alles, was eventuell hinter einem Leerzeichen (bzw. Tabulator-Zeichen) steht, nicht mehr mit eingelesen wird (es sei denn, dies ist beabsichtigt), und zum anderen findet keine Längenüberprüfung statt. Man ist nicht vor einer Bereichsüberschreitung (Pufferüberlauf) geschützt. Deswegen wird gewöhnlich zur Methode `getline` von `cin` gegriffen. Diese Funktion deckt beide Problemfälle ab:

```
// carray1.cpp
#include <iostream>
using namespace std;

int main(void) {
   char carray[80];
   cout << "Bitte Eingabe machen : ";
   cin.getline( carray, sizeof(carray) );
   cout << "Ihre Eingabe:\n" << carray << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
Bitte Eingabe machen : Solution to the problem!
Ihre Eingabe:
Solution to the problem!
```

2.4.3 C-Strings: Bibliotheksfunktionen

Falls Sie schon probiert haben (C-Programmierer wissen das sowieso schon), einem Array ein anderes Array (oder einen C-String) zuzuweisen, werden Sie feststellen, dass dies nicht funktioniert:

```
char carray1[80];
char carray2[] = "Ein C-String";
carray1 = carray2;          // Geht nicht - Fehler
carray1 = "Ein C-String"    // Geht auch nicht - Fehler
```

Bei den C-Strings können Sie dann auf die Standardfunktionen der C-Bibliothek zugreifen. In diesem Fall sind dies die Funktionen `strcpy()` und `strncpy()`, wobei `strcpy()` hier nicht behandelt wird, da es ohne Kontrolle der Länge die Gefahr eines Pufferüberlaufs erhöht. Die Syntax lautet:

```
#include <cstring> // in C: <string.h>
char*  strncpy ( char * dst, const char * src, size_t n );
```

Hiermit werden vom String, der in `src` steht, `n` Zeichen nach `dst` kopiert. Als Rückgabewert erhalten Sie einen Zeiger auf den Anfang von `dst` oder, im Falle eines Fehlers, `NULL`. Die Definition dieser Funktion finden Sie in der Header-Datei *<cstring>* (ohne *.h*). In C ist dies die Header-Datei *<string.h>*.

[»] Hinweis

Die C-Standardbibliotheks-Funktionen finden Sie natürlich weiterhin in den Header-Dateien *stdio.h*, *stdlib.h*, *string.h*, *math.h* usw. und können diese auch so verwenden. In C++ sind sie aber auch unter den Namen *cstdio*, *cstdlib*, *cstring*, *cmath* usw. vorhanden, nur mit dem Unterschied, dass der Inhalt im `std`-Namensbereich einkopiert ist. Mehr zum Namensbereich und zu `std` in Abschnitt 3.2, »Namensräume (Namespaces)«.

Hierzu ein Beispiel, wie Sie einen C-String in einen anderen kopieren können:

```
// carray2.cpp
#include <iostream>
#include <cstring>
using namespace std;

int main(void) {
   char carray1[80];
   char carray2[] = "Ein C-String";
   strncpy ( carray1, carray2, sizeof(carray1)-1 );
   cout << "carray1: " << carray1 << '\n';
   return 0;
}
```

Die Größe des C-Strings lassen wir uns mit dem `sizeof`-Operator –`1` ermitteln. Minus 1 deshalb, weil auch noch Platz für das Stringende-Zeichen benötigt wird.

Wollen Sie jetzt an den C-String einen weiteren C-String anhängen, können Sie leider nicht mehr `strncpy()` verwenden, da hiermit der alte Inhalt komplett überschrieben würde. Deshalb wird eine der C-Funktionen `strcat()` oder `strncat()` benötigt, wobei hier wiederum auf `strcat()` nicht eingegangen wird:

```
#include <cstring>
char* strncat ( char* dst, const char* src, size_t n );
```

Mit dieser Funktion hängen Sie `n` Zeichen des Strings `src` an das Ende von `dst`. Hierzu ein weiteres Beispiel, das das Aneinanderhängen von C-Strings demonstriert:

```
// carray3.cpp
#include <iostream>
#include <cstring>
```

```
using namespace std;

int main(void) {
   char name[60];
   char vname[30];
   char nname[30];

   cout << "Vorname  : ";
   cin.getline( vname, sizeof(vname) );
   cout << "Nachname : ";
   cin.getline( nname, sizeof(nname) );

   strncpy( name, vname, sizeof(name)-1 );
   strncat( name, " ", 1 );
   strncat( name, nname, sizeof(name)-strlen(name)-1);
   name[sizeof(name)-1] = '\0';

   cout << "Vollständiger Name: " << name << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vorname  : Jürgen
Nachname : Wolf
Vollständiger Name: Jürgen Wolf
```

Die folgende Zeile ist nötig, weil `strncat()` keine String-Terminierung vornimmt, wenn Sie den C-String komplett ausfüllen. Daher sollte der C-String zur Sicherheit erneut terminiert werden:

```
name[sizeof(name)-1] = '\0';
```

Bestimmt ist Ihnen in der Zeile

```
strncat( name, nname, sizeof(name)-strlen(name)-1);
```

die Subtraktion mit der Funktion `strlen()` und `name` als Parameter aufgefallen. Die Funktion `strlen()` zählt die Anzahl der Zeichen eines C-Strings bis zum Terminierungszeichen. Wenn Sie `strlen()` also auf einen nicht terminierten String anwenden, kommt es zu einem Programmfehler. Die Syntax von `strlen()` lautet:

```
#include <cstring>
size_t  strlen ( const char* string );
```

Neben diesen Funktionen gibt es noch eine Menge weiterer Funktionen für C-Strings, die Sie weiter unten aufgelistet finden. Zuvor aber noch ein paar (Grund-) Sätze zu den C-Strings.

Ihnen ist sicherlich aufgefallen, dass die Verwendung von C-Strings recht umständlich ist, und häufig kann eine falsche Verwendung zu gefährlichem Code, wie zum Beispiel Pufferüberläufen (Buffer Overflow), führen. Funktionen wie `strcpy()` oder `strcat()` (ohne `n` dazwischen) bieten keine Längenüberprüfung an. Somit kann eine solche Funktion, wenn man nicht richtig aufpasst, ungehindert in einen unerlaubten Speicherbereich schreiben.

Leider hat die Vergangenheit gezeigt, dass sich viele Programmierer einfach nicht an diese Regeln gehalten haben, so dass heute kein Tag vergeht, an dem nicht wieder eine Sicherheitslücke in einem Programm gefunden und gegebenenfalls ausgenutzt wurde. Gerade im Bereich Pufferüberlauf werden wohl die meisten »Fehler« gemacht. In einer Welt, in der jeder Computer Zugang zu einem Netzwerk hat, ist dies ein nicht mehr zu übersehendes Problem.

Und aus dem Grund, dass die C-Strings recht umständlich zu bedienen sind und der Sicherheitsaspekt häufig vernachlässigt worden ist, wurde in C++ die `string`-Klasse eingeführt, die erheblich einfacher zu verwenden ist, mehr Funktionalität bietet und vor allem erheblich sicherer ist. Aber wie bereits erwähnt – da es noch viele Zeilen C-Code gibt, müssen Sie sich als ernsthafter C++-Programmierer auch mit den C-Strings befassen. Und wenn die Ausführgeschwindigkeit auch noch von Bedeutung ist (aber nur dann), sind die C-Strings immer noch wesentlich effizienter.

Jetzt noch die restlichen C-Funktionen (siehe Tabelle 2.1), die Sie in Verbindung mit C-Strings verwenden können (alle Funktionen benötigen die Header-Datei <*cstring*>):

Funktion	Beschreibung
`char *strchr( const char *s, int ch );`	Diese Funktion gibt die Position im C-String `s` beim ersten Auftreten von `ch` zurück. Tritt das Zeichen `ch` nicht auf, wird `NULL` zurückgegeben.
`char *strrchr( const char *s, int ch );`	Diese Funktion ähnelt der Funktion `strchr()`, nur dass hier das erste Auftreten des Zeichens von hinten bzw. das letzte ermittelt wird.
`int strcmp( const char *s1, const char *s2 );`	Sind beide C-Strings identisch, gibt diese Funktion `0` zurück. Ist der C-String `s1` kleiner als `s2`, so ist der Rückgabewert kleiner als 0; und ist `s1` größer als `s2`, dann ist der Rückgabewert größer als 0.

Tabelle 2.1 Standard-C-Funktionen in <cstring> (bzw. in C <string.h>)

Funktion	Beschreibung
`int strncmp( const char *s1, const char *s2, size_t n );`	Hiermit werden die ersten `n` Zeichen von `s1` und die ersten `n` Zeichen von `s2` lexikografisch miteinander verglichen. Der Rückgabewert ist derselbe wie schon bei `strcmp()`.
`int strcspn( const char *s1, const char *s2 );`	Sobald ein Zeichen, das in `s2` angegeben wurde, im C-String `s1` vorkommt, liefert diese Funktion die Position dazu zurück.
`char *strstr( const char *s1, const char *s2 );`	Damit wird der C-String `s1` nach einem C-String mit der Teilfolge `s2` ohne `'\0'` durchsucht.
`char *strtok( char *s1, const char *s2 );`	Damit wird der C-String `s1` durch das Token getrennt, das sich in `s2` befindet. Ein Token ist hier ein C-String, der keine Zeichen aus `s2` enthält.
`void *memchr( const void *buffer, int c, size_t n);`	Diese Funktion sucht in den ersten `n` Bytes in `buffer` nach dem Zeichen `c`. Sollten Sie den ganzen String durchsuchen wollen, können Sie die Funktion `strchr()` verwenden. Tritt dabei ein Fehler auf oder wird das Zeichen nicht gefunden, gibt diese Funktion `NULL` zurück.
`int memcmp( const void *s1, const void *s2, size_t n);`	Mit `memcmp()` werden die ersten `n` Bytes im Puffer `s1` mit dem Puffer `s2` lexikografisch verglichen. Der Rückgabewert ist derselbe wie schon bei `strcmp()`.
`void *memcpy( void *dest, const void *src, size_t n );`	Mit der Funktion `memcpy()` können Sie `n` Bytes aus dem Puffer `src` in den Puffer `dest` kopieren. Die Funktion gibt die Anfangsadresse von `dest` zurück.
`void *memmove( void *dest, const void* src, size_t n );`	Die Funktion erfüllt denselben Zweck wie die Funktion `memcpy()`, mit einem einzigen, aber gravierenden Unterschied: `memmove()` stellt sicher, dass im Fall einer Überlappung der Speicherbereiche der Überlappungsbereich zuerst gelesen und dann überschrieben wird. Auch die Rückgabewerte sind bei `memmove()` dieselben wie bei `memcpy()`.
`void *memset( void *dest, int ch, unsigned int n );`	Mit dieser Funktion füllen Sie die ersten `n` Bytes der Adresse `dest` mit den Zeichen `ch` auf.

Tabelle 2.1 Standard-C-Funktionen in <cstring> (bzw. in C <string.h>) (Forts.)

Hinweis [«]

Mit den `mem...`-Funktionen können Sie ganze Speicherblöcke kopieren, vergleichen, initialisieren und durchsuchen.

2.5 Arrays und Zeiger

Arrays und Zeiger haben in C++ eine enge und gute Beziehung zueinander, aber sie sind nicht identisch oder ähnlich. Ein Array belegt nämlich zum Programmstart einen konstanten Speicher mit *n* Elementen, dessen Anfangsadresse nicht mehr verschoben werden kann. Man kann auch sagen, ein Array besitzt einen konstanten Zeiger auf das erste Element.

Einem Zeiger hingegen muss man erst einmal einen Wert zuweisen, damit dieser auch auf einen belegten Speicher zeigt. Außerdem kann der Wert eines Zeigers später nach Belieben einem anderen Wert (Speicherobjekt) zugewiesen werden. Ein Zeiger muss außerdem nicht nur auf den Anfang eines Speicherblocks zeigen. Hierzu ein einfaches Beispiel:

```
// zarray1.cpp
#include <iostream>
using namespace std;

int main(void) {
   int iarray[] = { 11, 22, 33 };
   // iptr zeigt auf das erste Element von iarray
   int* iptr = iarray;
   for(unsigned int i=0; i<sizeof(iarray)/sizeof(iarray[0]); i++ ) {
      cout << *iptr << '\n';
      iptr++;    // nächstes Element
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
11
22
33
```

Zunächst bestimmen Sie ein `int`-Array `iarray` mit drei Elementen. Anschließend weisen Sie dem Zeiger `iptr` die Adresse des ersten Elements zu.

In der folgenden Schleife wird der Wert mit Hilfe des Indirektionsoperators `*`, auf den `iptr` indirekt verweist, ausgegeben (im Beispiel `11`). Anschließend wird der Zeiger bzw. die Adresse inkrementiert. Da hier nicht der Indirektionsoperator verwendet wurde, bezieht sich diese Inkrementierung auf die Adresse des Zeigers. Und eine Erhöhung eines Zeigers um den Wert *n* hat denselben Effekt wie eine Erhöhung eines Array-Elements mit Hilfe des Indizierungsoperators um den Wert *n*. Somit wird also der Wert der Adresse nicht um eins erhöht, sondern um die Größe des Typs, die der Zeiger repräsentiert.

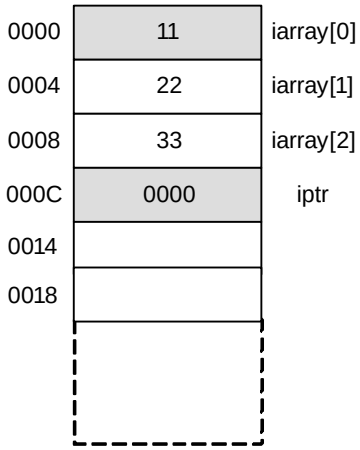

Abbildung 2.7 »iptr« verweist auf das erste Element von »iarray«

Im Beispiel eines Integers sind dies gewöhnlich vier Bytes (abhängig von der Architektur). Somit bedeutet also `Typ + n` in der Praxis:

```
Typ + sizeof(Typ) * n
```

Hätten Sie im Beispiel statt `iptr++` hier `(*iptr++)` mit dem Indirektionsoperator verwendet, hätten Sie tatsächlich den Wert des ersten Array-Elements um eins erhöht.

Nachdem also die Adresse von `iptr` inkrementiert wurde, ergibt sich folgendes Bild:

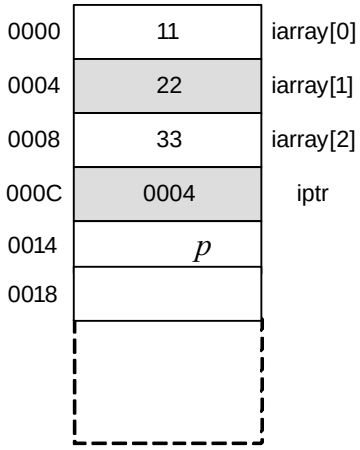

Abbildung 2.8 »iptr« verweist auf das zweite Element von »iarray«

Nach der Ausgabe wird die Adresse erneut um eins inkrementiert, wodurch der Zeiger `iptr` nun auf das letzte Element im Array zeigt, was auch wieder ausgegeben wurde. Die Abbruchbedingung der Schleife wurde mit

```
i < sizeof(iarray)/sizeof(iarray[0]);
```

angegeben. Die Schleife wird somit so lange durchlaufen, bis alle Elemente ausgegeben wurden (siehe auch Abschnitt 2.3.4, »Anzahl der Elemente eines Arrays ermitteln«).

Wie bereits in diesem Beispiel kurz erwähnt, ist es auch möglich, mit Hilfe eines Zeigers und des Indirektionsoperators indirekt den Wert der einzelnen Arrays zu verändern bzw. einen (neuen) Wert zuzuweisen – genau so, wie es schon in dem Abschnitt über Zeiger mit den Variablen gemacht wurde. Hier nun ein Beispiel dafür, wie Sie einem Array indirekt über einen Zeiger Werte zuweisen bzw. diese ändern:

```
// zarray2.cpp
#include <iostream>
using namespace std;

int main(void) {
   int iarray[5];
   // iptr zeigt auf das erste Element von iarray
   int* iptr = iarray;
   for(unsigned int i=0; i<sizeof(iarray)/sizeof(iarray[0]); i++ ) {
      cout << "Bitte Wert eingeben: ";
      cin >> *iptr;
      iptr++;    // nächstes Element
   }
   cout << "Die Werte lauten :\n";
   for(unsigned int i=0; i<sizeof(iarray)/sizeof(iarray[0]); i++ ) {
      cout << iarray[i] << '\n';
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Bitte Wert eingeben: 11
Bitte Wert eingeben: 22
Bitte Wert eingeben: 33
Bitte Wert eingeben: 44
Bitte Wert eingeben: 55
Die Werte lauten :
11
```

```
22
33
44
55
```

Die Eingabe von Werten mit Hilfe von

```
cin >> *iptr;
```

entspricht also der folgenden Eingabe:

```
cin >> iarray[i];
```

Sie wissen jetzt, dass man mit Zeigern auch auf ein beliebiges Element im Array verweisen kann. Wollen Sie zum Beispiel direkt mit dem Zeiger auf das dritte Element im Array zugreifen, so gehen Sie wie folgt vor:

```
// iptr zeigt auf das dritte Element von iarray
iptr = &iarray[2];
// Neuer Wert für das dritte Element im Array
*iptr = 1000;
```

Es gibt aber auch noch eine weitere Möglichkeit, um den Wert des dritten Arrays indirekt zu verändern:

```
// iptr zeigt auf das erste Element von iarray
int* iptr = iarray;
// Neuer Wert für das dritte Element im Array
*(iptr + 2) = 1000;
```

Damit der Zeiger tatsächlich auf die nächste Adresse zeigt, muss `iptr + 2` zwischen Klammern stehen, weil Klammern eine höhere Bindungskraft als der Dereferenzierungsoperator haben und somit zuerst ausgewertet werden. Sollten Sie die Klammern vergessen, würde nicht auf die nächste Adresse verwiesen, sondern auf den Wert, auf den der Zeiger `iptr` weist, und dieser würde dann um den Wert zwei erhöht.

Genau betrachtet, stellt der Indizierungsoperator `[]` eine Schreibweise für die Adressierung mit Hilfe der Zeigerarithmetik dar, wie das folgende Beispiel demonstrieren soll:

```
// zarray3.cpp
#include <iostream>
using namespace std;

int main(void) {
   int iarray[] = { 11, 22, 33 };
   // zweites Element ausgeben
```

```
    cout << iarray[1] << '\n';
    // als Zeigerarithmetik ...
    cout << *(iarray + 1) << '\n';
    // drittes Element ändern
    *(iarray + 2) = 44;
    // drittes Element ausgeben
    cout << iarray[2] << '\n';
    return 0;
}
```

Das Programm bei der Ausführung:

```
22
22
44
```

Beide Ausdrücke haben somit dieselbe Bedeutung:

```
iarray[2] = 0;
*(iarray + 2) = 0;
```

Im ersten Beispiel wird über den Indexoperator auf das dritte Element des Arrays `iarray` zugegriffen. Im zweiten Beispiel wird das Gleiche gemacht, nur durch eine Addition der Zahl 2 zur Adresse des ersten Elements des Arrays `iarray`. Die Klammerung hat hierbei eine höhere Bindungskraft als der Indirektionsoperator und wird daher zuerst ausgewertet. Erst daraufhin wird durch den Indirektionsoperator `*` der entsprechende Wert dereferenziert.

Um also auf das *n*-te Element eines Arrays zuzugreifen, haben Sie die folgenden (direkten und indirekten) Möglichkeiten:

```
int iarray[10];       // Deklaration
int *iptr1, *iptr2;
iptr1 = iarray;       // iptr1 auf Anfangsadresse von iarray
iptr2 = iarray + 3;   // iptr2 auf 4. Element von iarray

iarray[0]  = 99;      // Zuweisung an iarray[0]
iptr1[1]   = 88;      // Zuweisung an iarray[1]
*(iptr1+2) = 77;      // Zuweisung an iarray[2]
*iptr2     = 66;      // Zuweisung an iarray[3]
```

[»] Hinweis

Auch wenn Sie in der Praxis bei den Arrays den Indirektionsoperator `*` statt des Indizierungsoperators `[]` einsetzen können, ist davon abzuraten, da dies den Code nur unnötig verkompliziert. Auch in puncto Codeoptimierung lässt sich dadurch kein Vorteil erzielen.

2.5.1 C-Strings und Zeiger

Bei der Verwendung von C-Strings und Zeigern gibt es eigentlich nichts Neues zu berichten, was nicht auch für Arrays und Zeiger gilt. Letztendlich sind C-Strings nichts anderes als Arrays aus einzelnen Zeichen statt Zahlen. Somit gilt alles zuvor Beschriebene auch in Verbindung mit den C-Strings und Zeigern.

Aber hierzu vielleicht noch das Beispiel einer Subtraktion von Zeigern. Solch eine Subtraktion sollte allerdings nur dann verwendet werden, wenn (logischerweise) zwei Zeiger auf ein Element im selben Array bzw. C-String zeigen. Wenn Sie zwei Zeiger subtrahieren, erhalten Sie gewöhnlich die Anzahl der Array- bzw. C-String-Elemente, die zwischen den beiden Zeigern liegen:

```
// zarray4.cpp
#include <iostream>
using namespace std;

int main(void) {
   char carray[] = "Ein einfacher String";
   char *cptr1, *cptr2;
   cptr1 = carray;
   while( *cptr1 != ' ' ) {
      cptr1++;
   }
   cptr2 = ++cptr1;
   while( *cptr2 != ' ' ) {
      cptr2++;
   }
   cout << cptr2-cptr1 << '\n';
   return 0;
}
```

Die Ausgabe des Programms beträgt `9` – was bedeutet, dass zwischen der Adresse des Zeigers `cptr2` und des Zeigers `cptr1` 9 Elemente vom Typ `char` liegen. Im Beispiel wurde jeweils nach einem Leerzeichen gesucht und der Abstand von einem zum nächsten Leerzeichen ausgegeben.

2.5.2 Mehrfache Indirektion

Zugegeben, die Überschrift ist ein wenig verwirrend, und im Grunde will ich auch gar nicht allzu sehr auf dieses Thema eingehen, aber es ist auch möglich, Zeiger auf Zeiger zu deklarieren:

```
Typ** name;
```

Hier haben Sie einen Zeiger deklariert, der auf einen Zeiger verweist, der wiederum auf eine Variable verweist und somit auf diese Variable zugreifen kann. Das Ganze wird als *mehrfache Indirektion* bezeichnet. Theoretisch können Sie noch mehr als diese zweifache Indirektion verwenden:

```
Typ**** name;
```

Aber irgendwo ist die Grenze erreicht. Gewöhnlich werden Zeiger auf Zeiger mit zwei Indirektionsoperatoren verwendet. Das Haupteinsatzgebiet liegt im Allgemeinen bei der dynamischen Erzeugung von mehrdimensionalen Arrays (daher wurde dieses Thema auch bei den Arrays behandelt und nicht bei den Zeigern), wie dies zum Beispiel bei einer Matrizenberechnung benötigt wird.

In der »Praxis« lässt sich ein Zeiger auf einen Zeiger wie folgt einsetzen (wobei das Beispiel relativ wenig Sinn macht):

```
// ptrptr1.cpp
#include <iostream>
using namespace std;

int main(void) {
   int iwert = 111;
   // iptr verweist auf die Adresse von wert
   int *iptr=&iwert;
   // iptrptr verweist auf die Adresse von iptr
   int **iptrptr=&iptr;

   // Dereferenzierung
   cout << "*iptr     : " << *iptr << '\n';
   cout << "**iptrptr : " << **iptrptr << '\n';

   // doppelt indirekte Zuweisung ;-)
   **iptrptr = 222;
   // Dereferenzierung
   cout << "*iptr     : " << *iptr << '\n';
   cout << "**iptrptr : " << **iptrptr << '\n';

   // einfache indirekte Zuweisung
   *iptr =  333;
   // Dereferenzierung
   cout << "*iptr     : " << *iptr << '\n';
   cout << "**iptrptr : " << **iptrptr << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
*iptr     : 111
**iptrptr : 111
*iptr     : 222
**iptrptr : 222
*iptr     : 333
**iptrptr : 333
```

Hätten Sie im Beispiel statt der Dereferenzierung von `iptrptr` mit der doppelten Indirektion eine einfache Indirektion verwendet, so hätten Sie nicht den Wert der Variablen `iwert` verändert, sondern der Zeiger `iptrptr` würde auf eine ungültige Speicheradresse zeigen.

Somit lässt sich zusammenfassen (auf das Beispiel bezogen), dass `iptrptr` (ohne einen Indirektionsoperator) eine Variable vom Typ `char**` ist, die die Adresse von `iptr` enthält. `*iptrptr` (mit einem Indirektionsoperator) ist eine Variable vom Typ `char*`, die wiederum die Adresse von `iwert` enthält, und `**iptrptr` (mit zwei Indirektionsoperatoren) ist eine Variable vom Typ `char` mit dem eigentlichen Wert von `iwert` – also am Anfang `111`.

2.5.3 C-String-Tabellen

C-String-Tabellen sind den Zeigern auf Zeiger nicht unähnlich, aber sie sind – wie schon die Relation von »Array und Zeiger« – nicht dasselbe, sondern mehrdimensionale `char`-Arrays. Hier nun eine solche C-String-Tabelle:

```
char* cptrptr[] = { "String1", "String2", "String3" };
```

Von den Zeigern wissen Sie ja noch, dass die Schreibweisen `*ptr` und `ptr[0]` auf dieselbe Adresse verweisen. Dasselbe lässt sich auch über `**cptrptr` und `*cptrptr[0]` sagen.

Somit können Sie auch hier wie folgt auf die einzelnen Elemente zugreifen:

```
cout << *cptrptr << '\n';            // String1
cout << *(cptrptr + 1) << '\n';      // String2
cout << *(cptrptr + 2) << '\n';      // String3
```

Erneut sollten Sie der Lesbarkeit zuliebe den Indizierungsoperator `[]` verwenden:

```
cout << cptrptr[0] << '\n';          // String1
cout << cptrptr[1] << '\n';          // String2
cout << cptrptr[2] << '\n';          // String3
```

Natürlich können Sie ebenso auf die Zeichen der einzelnen C-Strings zugreifen. Bei den folgenden drei Beispielen greifen Sie automatisch auf den dritten Buchstaben des zweiten Strings zu. Drei verschiedene Möglichkeiten stehen Ihnen dafür zur Verfügung:

```
// 2.String "String2" -> 3. Buchstabe 'r'
cout << *(*(cptrptr+1)+2) << '\n';
cout << *(cptrptr[1]+2) << '\n';
cout << cptrptr[1][2] << '\n';
```

Tabelle 2.2 zeigt, wie Sie noch auf den *n*-ten C-String und das *m*-te Zeichen zugreifen können:

Zugriff auf ...	Möglichkeit 1	Möglichkeit 2	Möglichkeit 3
1.String, 1.Zeichen	`**marray`	`*marray[0]`	`marray[0][0]`
i.String, 1.Zeichen	`**(marray+i)`	`*marray[i]`	`marray[i][0]`
1.String, i.Zeichen	`*(*marray+i)`	`*(marray[0]+i)`	`marray[0][i]`
i.String, j.Zeichen	`*(*(marray+i)+j)`	`*(marray[i]+j)`	`marray[i][j]`

Tabelle 2.2 Zugriffsmöglichkeiten von Zeigern auf Zeiger und mehrdimensionale Arrays

Natürlich können Sie solche C-String-Tabellen auch dynamisch anlegen. Die Deklaration einer solchen Tabelle sieht wie folgt aus:

```
char *words[256];
```

Hiermit haben Sie ein `char`-Array mit 256 `char`-Zeigern deklariert. Sie können damit jedem dieser 256 Zeiger eine Adresse auf einem C-String zuweisen.

Das folgende Beispiel demonstriert nochmals den gesamten Vorgang zu den C-String-Tabellen in der Praxis:

```
// ptrptr2.cpp
#include <iostream>
using namespace std;

int main(void) {
   const char* cptrptr[] = { "String1", "String2", "String3" };
   const char* cstring[3];
   char carray[] = " und ";

   cout << *cptrptr << '\n';
   cout << *(cptrptr + 1) << '\n';
   cout << *(cptrptr + 2) << '\n';

   cout << cptrptr[0] << '\n';
```

```
   cout << cptrptr[1] << '\n';
   cout << cptrptr[2] << '\n';

   // 2.String "String2" -> 3. Buchstabe 'r'
   cout << *(*(cptrptr+1)+2) << '\n';
   cout << *(cptrptr[1]+2) << '\n';
   cout << cptrptr[1][2] << '\n';

   cstring[0] = cptrptr[0];
   cstring[1] = carray;
   cstring[2] = cptrptr[1];

   cout << cstring[0] << cstring[1] << cstring[2] << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
String1
String2
String3
String1
String2
String3
r
r
r
String1 und String2
```

Sicherlich sind Ihnen im Listing *ptrptr2.cpp* die beiden `const`-Initialisierungen von `cptrptr` und `cstring` wie folgt aufgefallen:

```
...
const char* cptrptr[] = { "String1", "String2", "String3" };
const char* cstring[3];
...
```

Ältere Compiler (eigentlich alle außer der `g++` ab der Version 4) würden diesen Code auch ohne Warnmeldung ohne die `const`-Initialisierung übersetzen. Neuere Compiler hingegen würden eine Warnung wie folgt ausgeben:

```
Warnung: veraltete Konvertierung von Zeichenkettenkonstante in "char*"
```

Zwar könnten Sie jetzt diese Warnmeldung mit Hilfe von Compiler-Flags (beispielsweise `-Wno-write-strings` beim `g++`) unterdrücken, aber dies wäre wohl nicht im Sinne des Erfinders. Daher gilt bei Zeigern, die mit einer Zeichenkette initialisiert werden, dass diese immer `const` sein sollen.

2.5.4 Arrays im Heap (dynamisches Array)

Häufig will man zum Programmstart so wenig Ressourcen wie nötig verbrauchen und die Speicherobjekte zur Laufzeit dynamisch anlegen. Wenn Ihnen das Reservieren von Speicher einzelner Variablen zur Laufzeit in Abschnitt 2.1.5, »Dynamisch Speicherobjekte anlegen und zerstören – 'new' und 'delete'«, etwas sinnlos erschienen ist, so dürfte Ihnen der jetzt vorliegende Grund sinnvoller erscheinen. Die Syntax, mit der Speicherplatz vom Heap für ein Array reserviert wird, lautet:

```
Zeiger = new Typ [Elemente];
```

Hier ist `Zeiger` das erste Objekt in einem Array von `Elemente`-Objekten. Ein Beispiel dazu:

```
int* iptr = new int [100];
```

Hiermit zeigt der Zeiger `iptr` auf das erste Objekt von 100 `int`-Objekten. Jetzt können Sie mit Hilfe des Indizierungsoperators Werte an das Array zuweisen.

```
iptr[0] = 100;
iptr[1] = 200;
...
iptr[99] = 9900;
```

Hierzu folgt nun ein Programmbeispiel, das die Abfrage ausführt, wie viel Speicherplatz Sie vom Heap reservieren wollen. Anschließend wird Platz für die von Ihnen angeforderte Menge reserviert, und Sie können Werte eingeben. Alle Werte werden addiert, so dass am Ende die Summe aller eingegebenen Werte ausgegeben wird. Zum Schluss wird der angeforderte Speicher vom Heap mit `delete` wieder freigegeben. Hierbei müssen Sie die eckigen Klammern mit angeben, um dem Compiler zu signalisieren, dass ein Array zu zerstören ist. Ohne die eckigen Klammern würden Sie nur das erste Element freigeben!

```
// dynarr1.cpp
#include <iostream>
using namespace std;

int main(void) {
   int *dynarray;
   int elements;
   int sum=0;
   cout << "Wie viele Werte wollen Sie speichern : ";
   cin >> elements;
   // Speicher reservieren
   dynarray = new int [elements];

   for(int i=0; i<elements; i++) {
```

```
      cout << i+1 << ".Wert : ";
      cin >> dynarray[i];
   }
   cout << "Die Summe aller Werte lautet : ";
   for(int i=0; i<elements; i++) {
      sum += dynarray[i];
   }
   cout << sum << '\n';
   // Speicher wieder freigeben
   delete [] dynarray;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Wie viele Werte wollen Sie speichern : 5
1.Wert : 11
2.Wert : 22
3.Wert : 33
4.Wert : 44
5.Wert : 55
Die Summe aller Werte lautet : 165
```

Hinweis **[«]**

Die Operatoren `new` und `delete` wurden bereits in Abschnitt 2.1.5, »Dynamisch Speicherobjekte anlegen und zerstören – 'new' und 'delete'«, behandelt.

Hier folgt wohl die unausweichliche Frage, was zu tun ist, wenn der Speicherplatz nicht ausreicht und man ein dynamisches Array weiter vergrößern will. Leider ist das nicht so einfach. Wer als C-Programmierer denkt, es gibt in C eine `realloc()`-Alternative wie zum Beispiel `renew`, der wird bitter enttäuscht werden. Wenn Sie ein bereits alloziertes Array vergrößern wollen, müssen Sie folgende Schritte ausführen:

- einen neuen, größeren Speicherplatz im Heap anfordern
- den Inhalt des alten Arrays ins neue Array kopieren
- das alte Array mit `delete` löschen
- den alten Zeiger des Arrays auf das neue Array verweisen lassen

Das Ganze könnte zum Beispiel in einer Schleife verpackt und endlos durchlaufen werden. Aber aus Gründen der Übersichtlichkeit soll dieser Vorgang gleich in eine Funktion verpackt werden, die als ersten Parameter das alte Array erwartet, als zweiten Parameter die Größe des alten Arrays und als dritten Parameter die Größe, die das neue (und alte) Array danach haben soll. Als Rückgabewert gibt

diese Funktion einen Zeiger auf die Anfangsadresse des neu reservierten Speichers im Heap zurück. In diesem Beispiel wurde außerdem die Übergabe von Arrays bzw. Zeigern an eine Funktion als Parameter vorgezogen, was in Abschnitt 2.6, »Parameterübergabe mit Zeigern, Arrays, Referenzen«, näher erläutert wird.

```
// dynarr2.cpp
#include <iostream>
using namespace std;

int* renew( int* old, int sizeold, int sizenew ) {
   // Größeren Speicher allozieren
   int* tmp = new int [sizenew];
   // Alten Inhalt nach tmp kopieren
   for( int i=0; i < sizeold; i++ ) {
      tmp[i] = old[i];
   }
   // Alten Inhalt löschen
   delete [] old;
   // Neues Array zurückgeben
   return tmp;
}

int main(void) {
   int *dynarray = new int[3];
   for( int i=0; i < 3; i++ ) {
      dynarray[i] = i;
   }
   // Array vergrößern auf fünf Elemente
   dynarray=renew( dynarray, 3, 5);
   for( int i=3; i < 5; i++ ) {
      dynarray[i] = i;
   }
   for( int i=0; i < 5; i++ ) {
      cout << dynarray[i] << '\n';
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
0
1
2
3
4
```

Zweidimensionale dynamische Arrays

Bei den Zeigern auf Zeiger wurde bereits erwähnt, dass sich hiermit mehrfach dimensionierte Arrays dynamisch erstellen lassen. Im Folgenden soll ein solches zweidimensionales Array mit *n* Zeilen und *m* Spalten erstellt werden. Aus

```
int** matrix;
```

soll somit

```
int matrix[zeile][spalte];
```

werden. Um ein solches zweidimensionales Array zu realisieren, müssen Sie zunächst Platz für die Zeilen oder genauer Zeilenadressen reservieren. Den Speicher für eine Zeile können Sie wie folgt reservieren:

```
int** matrix;
...
matrix = new int* [zeile];
```

Damit haben Sie schon einmal Platz für eine bestimmte Anzahl von Zeilen (bzw. für die erste Dimension) reserviert (hier drei Zeilen).

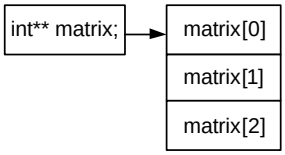

Abbildung 2.9 Reservierung des Speichers für die Zeile (erste Dimension)

Im nächsten Schritt können Sie für die einzelnen Zeilen (Zeile für Zeile) Speicherplatz für die Spalten (oder genauer für die zweite Dimension) reservieren. Dies wird gewöhnlich in einer Schleife realisiert, die Zeile für Zeile durchläuft:

```
for(int i = 0; i < zeile; i++) {
   matrix[i] = new int [spalte];
}
```

Jetzt haben Sie Speicherplatz für das zweidimensionale Array reserviert und können es mit Werten belegen (hier mit zwei Spalten).

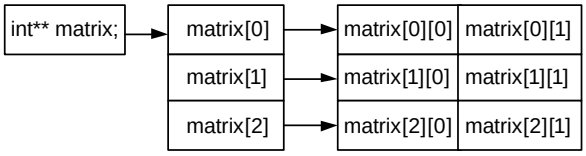

Abbildung 2.10 Nach der Reservierung des Speichers für die Spalte (3×2)

Beim Freigeben des Speichers mit `delete` müssen Sie allerdings darauf achten, dass dies in umgekehrter Reihenfolge geschieht. Hierzu nun das komplette Beispiel:

```
// dynarr3.cpp
#include <iostream>
using namespace std;

int main(void) {
   int i, j, zeile, spalte;
   int ** matrix;
   cout << "Wie viele Zeilen : ";
   cin >> zeile;
   cout << "Wie viele Spalten: ";
   cin >> spalte;

   // Speicher für die einzelnen Zeilen reservieren
   matrix = new int* [zeile];
   // Speicher für die einzelnen Spalten in der i-ten Zeile
   for(int i = 0; i < zeile; i++) {
      matrix[i] = new int [spalte];
   }
   // Mit beliebigen Werten initialisieren
   for (i = 0; i < zeile; i++) {
      for (j = 0; j < spalte; j++) {
         matrix[i][j] = i + j;   // matrix[zeile][spalte]
      }
   }
   // Inhalt ausgeben
   for (i = 0; i < zeile; i++) {
      for (j = 0; j < spalte; j++) {
         cout << matrix[i][j] << " ";
      }
      cout << '\n';
   }
   // Speicherplatz wieder freigeben
   // Wichtig! In umgekehrter Reihenfolge

   // Spalten der i-ten Zeile freigeben
   for(i = 0; i < zeile; i++)
      delete matrix[i];
   // Jetzt können die leeren Zeilen freigegeben werden
   delete [] matrix;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Wie viele Zeilen : 5
Wie viele Spalten: 4
0 1 2 3
1 2 3 4
2 3 4 5
3 4 5 6
4 5 6 7
```

2.6 Parameterübergabe mit Zeigern, Arrays und Referenzen

Sie kennen bereits die Möglichkeit, Daten zwischen Funktionen auszutauschen durch die Parameterübergabe und durch globale Variablen. Globale Variablen sollten aber aufgrund der Fehleranfälligkeit möglichst vermieden werden und sind in der Praxis selten sinnvoll.

In den Abschnitten zu den Funktionen haben Sie Parameter auch als formale Parameter kennengelernt. Solche formalen Parameter dienen immer als Platzhalter für die aktuellen Parameter.

2.6.1 Call by value

Bei der Übergabe von Parametern unterscheidet man zwischen zwei Arten, dem *call by value* und dem *call by reference*. Sie haben bisher im Kapitel über Funktionen die Werte nach der Art *call by value* an die Funktionen übergeben.

```
// callbyval.cpp
#include <iostream>
using namespace std;

void callbyvalue( int a, int b );

int main(void) {
   int iwert1 = 100, iwert2 = 5;
   callbyvalue( iwert1, iwert2 );
   cout << iwert1 << ' ' << iwert2 << '\n';
   return 0;
}

void callbyvalue( int a, int b ) {
   cout << a << " * " << b << " = " << (a*b) << '\n';
```

```
    a = 0;
    b = 0;
}
```

Das Programm bei der Ausführung:

```
100 * 5 = 500
100 5
```

Am Ende der Funktion `call by value` wurden die beiden übergebenen Werte auf `0` gesetzt, was auf die Originalwerte in der `main`-Funktion keinen Einfluss hat, wie die Ausgabe bestätigt. Somit können Sie also sicher sein, dass beim `call by value` immer nur eine Kopie an die Funktion übergeben wird.

Bei einer Übergabe als Kopie muss bei jedem Funktionsaufruf eine Kopie des oder der aktuellen Parameter erstellt werden. Dies ist bei einfachen Basisdatentypen kein Problem. Aber wenn die Objekte komplexer und umfangreicher werden – was in der objektorientierten Programmierung gewöhnlich der Fall ist –, müssen auch diese an die Funktion übergeben werden. Muss bei jedem Funktionsaufruf ein solches Objekt erzeugt und wieder zerstört werden, so kann sich dies auf das Laufzeitverhalten des Programms sehr negativ auswirken.

2.6.2 Call by reference – Zeiger als Funktionsparameter

Um solche Schwierigkeiten zu vermeiden, wird das *call by reference*-Verfahren bevorzugt. Dabei geben Sie nicht die Objekte selbst an eine Funktion, sondern nur die Adressen der Objekte. Dadurch können die Objekte durch eine Dereferenzierung des Zeigers direkt in der Funktion bearbeitet oder verändert werden. Natürlich bedeutet die Übergabe einer Adresse als Parameter auch, dass sich jede Veränderung (durch Dereferenzierung) auf das Original bezieht, weil keine Kopie mehr übergeben wird.

Sie müssen auch die Deklaration und Definition der Funktion anpassen, und beim Aufruf der Funktion müssen Sie selbstverständlich die Adresse und nicht mehr die Variable selbst übergeben.

```
void callbyreference( int* a, int* b );

// Funktionsaufruf
callbyreference( &iwert1, &iwert2 );
```

Wenn Sie den Wert auch verwenden wollen, ist zu beachten, dass Sie diesen in der Funktion mit dem Indirektionsoperator einsetzen müssen. Obwohl das selbstverständlich sein sollte, führt es immer wieder zu Fehlern beim *call by reference*-Verfahren:

```
// callbyref.cpp
#include <iostream>
using namespace std;

void callbyvalue( int a, int b );
void callbyreference( int* a, int* b );

int main(void) {
   int iwert1 = 100, iwert2 = 5;
   cout << "call by value:\n";
   callbyvalue( iwert1, iwert2 );
   cout << iwert1 << ' ' << iwert2 << "\n\n";
   cout << "call by reference:\n";
   callbyreference( &iwert1, &iwert2 );
   cout << iwert1 << ' ' << iwert2 << "\n\n";
   return 0;
}

void callbyvalue( int a, int b ) {
   cout << a << " * " << b << " = " << (a*b) << '\n';
   a = 0;

   b = 0;
}

void callbyreference( int* a, int* b ) {
   cout << *a << " * " << *b << " = "
        << (*a) * (*b) << '\n';
   *a = 0;
   *b = 0;
}
```

Das Programm bei der Ausführung:

```
call by value:
100 * 5 = 500
100 5

call by reference:
100 * 5 = 500
0 0
```

Anhand der Ausgabe des *call by reference*-Verfahrens lässt sich erkennen, dass sich eine Veränderung der übergebenen Adresse gleich auf das Original bezieht. Die Klammerungen in der Funktion `call by reference` wurden nur wegen der Übersichtlichkeit verwendet.

2.6.3 Call by reference mit Referenzen nachbilden

Vielleicht finden Sie die mehrfache Dereferenzierung mit dem Indirektionsoperator beim *call by reference*-Verfahren in den Funktionen auch etwas umständlich. Außerdem ist die Anwendung nicht sehr lesefreundlich und recht anfällig für Fehler. Dann kommt noch die umständliche Übergabe der Werte mit dem Adressoperator hinzu – viele kleine Dinge, die zum einen mehr Arbeit machen, und zum anderen zu Fehlern führen können.

Dieser Auffassung waren auch die Erfinder von C++, und sie haben vorwiegend deshalb die Referenzen (siehe Abschnitt 2.2, »Referenzen«) eingeführt. Mit den Referenzen können Sie das *call by reference*-Verfahren nachbilden. Das bedeutet: kein Adressoperator beim Funktionsaufruf und keine wilden Dereferenzierungen mehr in den Funktionen selbst und trotzdem alle Vorzüge des *call by reference*-Verfahrens. Es ist alles wie beim *call by value*-Verfahren, nur dass der Funktionskopf bei der Deklaration und Definition anders aussieht:

```
void callbyreference2( int& a, int& b );
```

Hierzu die nachgebildete `call by reference`-Funktion mit Referenzen in der Praxis:

```
// callbyref2.cpp
#include <iostream>
using namespace std;

void callbyreference2( int& a, int& b );

int main(void) {
   int iwert1 = 100, iwert2 = 5;
   cout << "call by reference (über Referenzen):\n";
   callbyreference2( iwert1, iwert2 );
   cout << iwert1 << ' ' << iwert2 << "\n\n";
   return 0;
}

void callbyreference2( int& a, int& b ) {
   cout << a << " * " << b << " = " << (a*b) << '\n';
   a = 0;
   b = 0;
}
```

Das Programm bei der Ausführung:

```
call by reference (über Referenzen):
100 * 5 = 500
0 0
```

Wollen Sie an die Funktion `callbyreference2` statt zweier einfacher Variablen zwei Zeiger übergeben, müssen Sie den Indirektionsoperator beim Funktionsaufruf verwenden:

```
int ival1 = 100, ival2 = 5;
int *iwert1 = &ival1, *iwert2 = &ival2;
callbyreference2( *iwert1, *iwert2 );
```

2.6.4 Arrays als Funktionsparameter

Arrays werden an Funktionen als Parameter wie Zeiger im *call by reference*-Verfahren übergeben. Zum Glück, denn man stelle sich vor, dass ein Array mit 1000 Objekten an eine Funktion kopiert wird. Die Deklaration und Definition des Funktionskopfs kann demnach so

```
void callarray( int* array );
```

oder auch so aussehen:

```
void callarray( int array[] );
```

Sie können beim Array also die Array-Deklaration mit der Zeiger-Deklaration als formalem Parameter einfach vertauschen. Beachten Sie allerdings, dass dies eine Besonderheit ist, denn bei Funktionen »zerfällt« ein Array sofort in einen Zeiger, und somit wird niemals wirklich ein Array an eine Funktion übergeben. Dieser Umstand ist für Anfänger etwas verwirrend, weil dadurch der Eindruck entsteht, dass Zeiger und Arrays dasselbe sind. Aber diese Umwandlung eines Arrays in einen Zeiger gilt wirklich nur für die formalen Parameter einer Funktion.

Dass ein Array hier in einen Zeiger zerfällt, macht es außerdem unmöglich anzugeben, wie viele Elemente das Array enthält. Folgendes wäre also falsch:

```
void callarray( int array[10] );   // falsch!!!
```

Daher ist es empfehlenswert, die Elemente eines Arrays immer als Extra-Argument mit anzugeben:

```
void callarray( int array[], int elements );
// oder auch
void callarray( int* array, int elements );
```

Aufgerufen wird diese Funktion wie folgt:

```
callarray( iarray, sizeof(iarray)/sizeof(iarray[0]));
```

Oder aber auch mit

```
callarray( &iarray[0], sizeof(iarray)/sizeof(iarray[0]));
```

Theoretisch müssen Sie hierbei nicht zwangsläufig die Adresse des ersten Array-Elements übergeben. Würden Sie `&iarray[2]` als Argument verwenden, würden Sie die Adresse des dritten Elements an die Funktion übergeben. Beachten Sie allerdings dann den zweiten Parameter. Denn sollten Sie alle Elemente durchlaufen, kann es passieren, dass Sie den Speicherbereich überschreiten. Hierzu nun das Programmbeispiel, wie Sie ein Array an eine Funktion als Parameter übergeben:

```
// callarray.cpp
#include <iostream>
using namespace std;

void callarray( int* array, int elements );

int main(void) {
   int iarray[] = { 11, 22, 33 };
   callarray( iarray, sizeof(iarray)/sizeof(iarray[0]));
   return 0;
}

void callarray( int* array, int elements ) {
   for( int i=0; i < elements; i++) {
      cout << array[i] << '\n';
   }
}
```

Das Programm bei der Ausführung:

```
11
22
33
```

Natürlich gilt auch hier, dass sich jede Veränderung des Arrays in der Funktion auch auf das Original bezieht, da ja auch mit Adressen gearbeitet wird.

Wenn Sie unbedingt ein Array als Kopie an eine Funktion übergeben wollen, können Sie sich eines Tricks bedienen, indem Sie ein Array in eine Struktur verpacken (siehe Abschnitt 2.8.1, »Strukturen«). Zu Referenzzwecken hierzu das Listing (ohne an dieser Stelle näher darauf einzugehen):

```
// callarray2.cpp
#include <iostream>
using namespace std;

struct sarray { int i[3]; };
typedef struct sarray SARRAY;
void callarray( SARRAY array, int elements );
```

```
int main(void) {
   SARRAY iarray;
   iarray.i[0] = 11;
   iarray.i[1] = 22;
   iarray.i[2] = 33;
   callarray(iarray, sizeof(iarray.i)/sizeof(iarray.i[0]));
   return 0;
}

void callarray( SARRAY array, int elements ) {
   for( int i=0; i < elements; i++) {
      cout << array.i[i] << '\n';
   }
}
```

2.6.5 Mehrdimensionale Arrays an Funktionen übergeben

Natürlich ist es auch möglich, mehrdimensionale Arrays an eine Funktion als Parameter zu übergeben. Wenn man ein zweidimensionales Array an eine Funktion übergibt, »zerfällt« es (Array auf Array) in einen Zeiger auf Arrays (nicht etwa in einen Zeiger auf einen Zeiger). Somit kann der Funktionskopf wie folgt deklariert und definiert werden:

```
void callmarray( int array[][elements] );
```

Oder, was auch der Compiler daraus machen würde:

```
void callmarray( int array(*ptr)[elements] );
```

Wie schon bei den einfachen Arrays als Parameter stellt die aufgerufene Funktion keinen Speicher für ein Array in der ersten Dimension bereit, was Sie gegebenenfalls selbst übernehmen müssen. Die zweite Dimension (und alle weiteren) müssen Sie immer mit angeben.

Hierzu wieder ein einfaches Beispiel, das demonstriert, wie Sie ein zweidimensionales Array an eine Funktion übergeben können:

```
// callmarray.cpp
#include <iostream>
using namespace std;

const int DIM1=4;
const int DIM2=3;

void callmarray( int marray[][DIM2], int dim1 );
```

```
int main(void) {
   int imarray[DIM1][DIM2];
   for(int i=0; i < DIM1; i++) {
      for(int j=0; j < DIM2; j++) {
         imarray[i][j] = i+j;
      }
   }
   callmarray(imarray, DIM1);
   return 0;
}

void callmarray( int marray[][DIM2], int dim1) {
   for(int i=0; i < dim1; i++) {
      for(int j=0; j < DIM2; j++) {
         cout << marray[i][j] << ' ';
      }
      cout << '\n';
   }
}
```

Das Programm bei der Ausführung:

```
0 1 2
1 2 3
2 3 4
3 4 5
```

2.6.6 Argumente an die main-Funktion übergeben

Auch der `main`-Funktion können beim Programmstart Parameter übergeben werden. Standardmäßig hat `main()` zwei Parameter:

```
int main( int argc, char **argv );
```

`argc` gibt die Anzahl der Argumente zurück, die der `main`-Funktion übergeben wurden. Da der Name des Programms automatisch als erstes Argument verwendet wird, ist dieser Wert immer mindestens mit `1` belegt. `argv` hingegen ist eine String-Tabelle, die die einzelnen Argumente als C-Strings beinhaltet. Beispielsweise enthält `argv` gewöhnlich folgende Einträge:

```
argv[0]    = Programmname
argv[1]    = erstes Argument
argv[2]    = zweites Argument
...
argv[argc] = 0
```

Das letzte Element in `argv` enthält also immer den Wert `0` und kann daher ebenso als Abbruchbedingung verwendet werden wie `argc`.

Hinweis [«]

Die Namen `argc` und `argv` sind so nicht vorgeschrieben, es können genauso gut andere sein. Aber in der Praxis werden fast immer diese Namen verwendet.

Dem Programm werden Argumente beim Aufruf übergeben. Unter MS Windows kann dabei das alte MS-DOS verwendet werden. Die MS-DOS-Eingabeaufforderung von MS Windows ist dafür aber auch geeignet. Unter Linux genügt eine einfache Konsole bzw. Shell. Die einzelnen Argumente, die dem Programm per Kommandozeile übergeben werden, müssen immer mit mindestens einem Leerzeichen getrennt sein. Ein Beispiel dazu:

```
Prompt > programmname argument1 argument2 argument3
argv[0] = programmname
argv[1] = argumente1
argv[2] = argumente2
argv[3] = argumente3
```

Sofern Sie eine Entwicklungsumgebung (IDE) verwenden, müssen Sie Kommandozeilen-Argumente anders übergeben. Viele Entwicklungsumgebungen bieten hierfür im Menü Ausführen noch ein Untermenü Parameter oder Ähnliches (abhängig von der IDE) an, um die Argumente noch vor dem Programmstart festzulegen.

Hinweis [«]

Wie Sie Kommandozeilen-Argumente bei einer Entwicklungsumgebung an das auszuführende Programm übergeben können, wird auf der Buch-CD für gängige Programme (bspw. MS VC++ 2008, Code::Blocks, KDevelop, Anjuta) beschrieben.

Hierzu ein Programmbeispiel, das die Argumente auswertet, die Sie dem Programm beim Start mit übergeben:

```
// argmain.cpp
#include <iostream>
using namespace std;

int main( int argc, char *argv[] ) {
   cout << "Die Argumente der Kommandozeile an main:\n";
   cout << "Programmname: " << argv[0] << '\n';
   for(int i=1; i < argc; i++) {
      cout << "argv[" << i << "]: " << argv[i] << '\n';
   }
```

```
   return 0;
}
```

Das Programm bei der Ausführung:

```
Prompt > argmain Hallo Welt wie gehts
Die Argumente der Kommandozeile an main:
Programmname: C:\Dev-Cpp\argmain.exe
argv[1]: Hallo
argv[2]: Welt
argv[3]: wie
argv[4]: gehts
```

In `argv[0]` befindet sich meistens der Programmname, dies muss aber nicht zwangsläufig so sein. Ein Beispiel dazu:

```
char *argv_for_new_app[] = {
   "ganzAndererName", ....argumente
};
char *application = "programmname";
execve(application, argv_for_new_app, envp);
```

Somit ist in `argv[0]` von `programmname` nun `ganzAndererName` zu lesen. Das ist u. a. ein effektiver Workaround für DOS/Windows-Plattformen, die keine symbolischen Links haben (d. h., manche Programme erkennen ihre Funktion an `argv[0]`).

2.7 Rückgabewerte von Zeigern, Arrays und Referenzen

Bei einfachen Basisdatentypen ist es nicht unbedingt nötig, für den Rückgabewert einen Zeiger oder Referenzen zu verwenden. Aber bei etwas umfangreicheren oder dynamisch erzeugten Objekten sollten Zeiger oder Referenzen verwendet werden.

2.7.1 Zeiger als Rückgabewert

Sofern der Rückgabewert ein bereits bestehendes Element ist, empfiehlt es sich, Referenzen statt Zeiger zu verwenden. Bei neuen (dynamisch erzeugten) Elementen sollte man auf einen Zeiger zurückgreifen. Die Syntax eines solchen Funktionskopfs sieht wie folgt aus:

```
Typ* function( parameter );
```

Aufgerufen wird eine solche Funktion folgendermaßen:

```
Typ* ptr;
...
ptr = function ( argumente );
```

Die Verwendung von Zeigern als Rückgabewert aus Funktionen wird gerne bei Arrays oder Strukturen eingesetzt. Gerade bei Arrays bzw. C-Strings ist (bzw. war) dies die einzige Möglichkeit, ganze Felder bzw. Zeichenketten aus einer Funktion zurückzugeben. Natürlich wird auch hier immer nur eine Adresse auf das erste Element zurückgegeben. Hierzu ein einfaches Beispiel:

```
// dynamicarray.cpp
#include <iostream>
using namespace std;

int* dynamic_array( int elements );
int* renew( int* old, int sizeold, int sizenew );

int main(void) {
   // Reserviert Speicher für zehn Elemente
   int *iptr = dynamic_array( 10 );
   // Wert an die einzelnen Elemente zuweisen
   for( int i=0; i < 10; i++ ) {
      iptr[i] = i;
   }
   // ... weitere fünf Elemente reservieren
   iptr = renew( iptr, 10, 15 );
   // ... und initialisieren
   for( int i=10; i < 15; i++ ) {
      iptr[i] = i;
   }
   // Alles ausgeben
   for( int i=0; i < 15; i++ ) {
      cout << iptr[i] << '\n';
   }
   // Speicher wieder freigeben
   delete [] iptr;
   return 0;
}

int *dynamic_array( int elements ) {
   int *array = new int [elements];
   // Anfangsadresse zurückgeben
   return array;
}
```

```
int* renew( int* old, int sizeold, int sizenew ) {
   // Größeren Speicher allozieren
   int* tmp = new int [sizenew];
   // Alten Inhalt nach tmp kopieren
   for( int i=0; i < sizeold; i++ ) {
      tmp[i] = old[i];
   }
   // Alten Inhalt löschen
   delete [] old;
   // Anfangsadresse zurückgeben
   return tmp;
}
```

Beide Funktionen, `dynamic_array` und `renew`, allozieren jeweils Speicherplatz auf dem Heap und geben die Anfangsadresse des Speicherblocks an den Aufrufer zurück.

Leider sind es gerade Funktionen, die Zeiger oder Referenzen zurückgeben, die häufig Fehlerquellen enthalten. Sehen Sie sich dazu folgendes Beispiel an:

```
// badarray.cpp
#include <iostream>
using namespace std;

int *a_ten_array( void );

int main(void) {
   int *iptr = a_ten_array(  );
   for( int i=0; i < 10; i++ ) {
      iptr[i] = i;
   }
   for( int i=0; i < 10; i++ ) {
      cout << iptr[i] << '\n';
   }
   return 0;
}

int *a_ten_array( void ) {
   int array[10] = { 0 };
   return array;
}
```

Viele Compiler geben schon eine Warnung bei der Übersetzung aus, dass eine Variable mit einer lokalen Adresse zurückgegeben wird. Bei mir macht das Programm folgende Ausgabe:

```
0
2293424
2
3
1
4469696
6
7
5456754
9
```

Nicht ganz das gewünschte Ergebnis. Wenn Sie sich noch an die Beschreibung des Stacks bei den Funktionen (siehe Abschnitt 1.10, »Funktionen«) erinnern können, wissen Sie, dass alle benötigten Daten einer Funktion (Parameter, lokale Variablen, Rücksprungadresse) beim Funktionsaufruf auf diesem angelegt werden. Die Daten bleiben so lange erhalten, bis sich die Funktion wieder beendet.

Die Funktion `a_ten_array()` gibt einen solchen lokalen Speicherbereich zurück – was somit ein ungültiger Speicherbereich ist. Dasselbe Problem tritt auch auf, wenn Sie eine Referenz auf ein lokales Objekt zurückgeben wollen.

Wenn Sie etwas von einer Funktion zurückgeben lassen wollen, haben Sie zwei Möglichkeiten:

- einen (mit `new`) dynamisch reservierten Speicher vom Heap
- einen statischen Puffer (`static`) – siehe Abschnitt 3.4.3, »Speicherklasse 'static'«

Hier die beiden Möglichkeiten, bezogen auf das Beispiel `a_ten_array()`:

```
// 1. Speicher vom Heap zurückgeben
int* a_ten_array1( void ) {
   int* array = new int [10];
   return array;
}

// 2. Einen Static-Speicherbereich zurückgeben
int* a_ten_array2( void ) {
   static int array[10];
   return array;
}
```

Die Verwendung eines statischen Puffers funktioniert allerdings nicht mehr, wenn eine Funktion rekursiv aufgerufen wird.

2.7.2 Referenz als Rückgabewert

Auch Referenzen können als Rückgabewert eingesetzt werden. Damit können Sie praktisch einen Funktionsaufruf wie ein Objekt verwenden – also so, als würden Sie direkt auf eine Variable zugreifen. Haben Sie zum Beispiel folgenden Prototyp einer Funktion

```
int& test_referenz( void );
```

so stellt jeder Funktionsaufruf von `test_referenz()` eine `int`-Variable dar. Somit bedeutet ein Aufruf wie

```
++test_referenz();
```

dass eine referenzierte Variable inkrementiert wird. Das Hauptanwendungsgebiet ist hierbei das Überladen von Operatoren – ein Thema, das allerdings erst später im Buch (siehe Abschnitt 4.5, »Operatoren überladen«) behandelt wird. So eine Operatorüberladung haben Sie mit den Operatoren `<<` und `cout` schon mehrmals verwendet. Zum Beispiel ist der Ausdruck

```
cout << "Ein schöner Tag";
```

eine Referenz auf das Objekt `cout`, was bedeutet, dass er selbst wieder das Objekt `cout` darstellt. Daher können Sie mit dem folgenden Ausdruck den Operator `<<` erneut anwenden:

```
cout << "Ein schöner Tag" << "ist heute\n";
```

Aber jede weitere Erklärung würde den Anfänger jetzt überfordern. Dies wird alles erst im Abschnitt zur Überladung von Operatoren behandelt (siehe Abschnitt 4.5).

Hierzu noch ein Beispiel, wo der Funktionsaufruf eine Referenz auf ein `int`-Objekt darstellt bzw. wie eine `int`-Variable verwendet werden kann.

```
// returnref.cpp
#include <iostream>
using namespace std;

int& test_referenz( void );

int main(void) {
   int* ptr;
   int wert;
   // Wert von "iwert" an "wert" zuweisen
   wert = test_referenz();
   // "iptr" auf "iwert" zeigen lassen
   iptr = &test_referenz();
```

```
   cout << "iptr : " << *iptr << '\n';
   cout << "wert : " << wert << '\n';

   // ... entspricht ++iwert
   ++test_referenz();
   cout << "iptr : " << *iptr << '\n';
   cout << "wert : " << wert << '\n';

   // iwert verdoppeln
   test_referenz() = test_referenz() * 2;

   cout << "iptr : " << *iptr << '\n';
   cout << "wert : " << wert << '\n';
   return 0;
}
int& test_referenz( void ) {
   static int iwert = 10;
   return iwert;
}
```

Das Programm bei der Ausführung:

```
iptr : 10
wert : 10
iptr : 11
wert : 10
iptr : 22
wert : 10
```

2.7.3 const-Zeiger als Rückgabewert

Sie haben gesehen, wie man auch Zeiger als Rückgabewerte verwenden kann. Wollen Sie verhindern, dass ein Aufrufer mit Hilfe dieses Zeigers die Daten verändert, so können Sie einen `const`-Zeiger verwenden. Dann kann der adressierte Speicherbereich von der aufrufenden Funktion nur gelesen werden. Die Syntax eines solchen Prototyps lautet:

```
const typ* function (parameter);
```

2.7.4 Array als Rückgabewert

Ein Array selbst können Sie nicht als Rückgabewert einer Funktion verwenden. Hier können Sie entweder auf Zeiger zurückgreifen (unter Berücksichtigung des Gültigkeitsbereichs), oder Sie verwenden denselben Trick wie schon bei der Übergabe eines Arrays als Parameter *by value* (also als Kopie). Sie packen das

Array in eine Struktur (siehe Abschnitt 2.8.1, »Strukturen«) und geben diese dann so verpackt an den Aufrufer zurück. Das Programmbeispiel dazu:

```
// retarray.cpp
#include <iostream>
using namespace std;

struct array { int wert[3]; };
struct array init_array(void);

int main(void) {
   struct array newarray = init_array();
   for(unsigned int i=0; i < sizeof(struct array)/sizeof(int); i++) {
      cout  << newarray.wert[i] << '\n';
   }
   return 0;
}

struct array init_array(void) {
   struct array arr;
   for(unsigned int i=0; i < sizeof(struct array)/sizeof(int); i++) {
      arr.wert[i] = i;
   }
   return arr;
}
```

2.7.5 Mehrere Rückgabewerte

Es ist standardmäßig nicht möglich, mehrere Werte auf einmal zurückgeben zu lassen. Auch hier können Sie mehrere Werte in einer Struktur verpacken und diese anschließend an den Aufrufer zurückgeben. Aus Effizienzgründen sollte man allerdings eine Adresse auf diese Struktur zurückliefern lassen.

Eine weitere Möglichkeit ist die Verwendung von Zeigern bzw. eine Parameterübergabe *by reference*. Sie wissen ja noch, dass sich hierbei eine Veränderung des Wertes tatsächlich auch auf das Original bezieht, da mit derselben Adresse gearbeitet wird. Ein Beispiel, worauf das hinausläuft:

```
// retmisc.cpp
#include <iostream>
using namespace std;

int ret_more( int& a, float& b );
```

```
int main(void) {
   int ret1, ret2;
   float ret3;
   // Funktionsaufruf
   ret1 = ret_more( ret2, ret3 );
   cout << "Die Ergebnisse der Berechnungen lauten :\n";
   cout << "Rechnung 1 : " << ret1 << '\n';
   cout << "Rechnung 2 : " << ret2 << '\n';
   cout << "Rechnung 3 : " << ret3 << '\n';
   return 0;
}

int ret_more( int& a, float& b ) {
   // Wichtiger Code hier ... bspw. mehrere Berechnungen
   a = 10;
   b = 11.11;
   return ( a * a);
}
```

Das Programm bei der Ausführung:

```
Die Ergebnisse der Berechnungen lauten :
Rechnung 1 : 100
Rechnung 2 : 10
Rechnung 3 : 11.11
```

In diesem Beispiel erhalten Sie drei »Rückgabewerte« – im Grunde zwar nur einen, aber durch das Nachbilden des *call by reference* mit Referenzen werden auch die anderen Werte »bearbeitet«.

2.8 Fortgeschrittene Typen

Da Sie jetzt alle Basisdatentypen in C++ kennen, ist es an der Zeit, die Sprache um »eigene« Typen zu erweitern. Statt von fortgeschrittenen oder erweiterten Typen könnte man auch von strukturierten Typen sprechen. Das Prinzip ist ähnlich wie bei den Arrays. Nur dass Sie jetzt statt einer Zusammenfassung des gleichen Typs Elemente beliebiger Typen zusammenfassen können. C++ kennt verschiedene solcher fortgeschrittenen Typen. Neben den in diesen Abschnitten beschriebenen Typen von Strukturen (`struct`), Unions (`union`) und Aufzählungen (`enum`) sind eigentlich die Klassen (`class`) das zentrale Thema dazu in der C++-Programmierung. Deshalb wird in Kapitel 4, »Objektorientierte Programmierung«, auch gesondert darauf eingegangen. Hier werden zunächst die Strukturen, Unions und Aufzählungen behandelt.

2.8.1 Strukturen

Mit den Strukturen haben Sie jetzt die Möglichkeit, mehrere Typen zu einem neuen Typ zusammenzufassen. Die Syntax einer solchen Zusammenfassung mit einer Struktur sieht wie folgt aus:

```
struct strukturName {
   Typ1;
   Typ2;
   ...
   TypN;
} VariablenBezeichner;
```

Eingeleitet mit dem Schlüsselwort `struct`, werden all diese Daten einer Struktur unter dem Namen `strukturName` zusammengefasst. Der Inhalt dieser Struktur (auch *Strukturmitglieder* genannt) wird nun in den geschweiften Klammern zusammengefasst. Am Ende können Sie optional einen Variablenbezeichner dieser Struktur deklarieren, mit dem Sie anschließend auf die einzelnen Strukturmitglieder zugreifen können. Und wie es auch schon bei der Deklaration von Variablen der Fall war, müssen Sie den erweiterten Strukturtyp ebenfalls mit einem Semikolon abschließen.

Strukturen deklarieren

Hierzu die Deklaration einer einfachen Zusammenfassung von Artikeln (was natürlich erheblich erweitert werden kann), wie sie bei diversen Programmen zur Verwaltung von Produkten verwendet werden kann.

```
struct artikel {
   int sachnummer;
   char bezeichnung[100];
   int anzahl;
};
```

Hiermit haben Sie einen erweiterten Typ namens `artikel` erstellt, der die Daten `sachnummer` vom Typ `int`, `bezeichnung`, einen C-String mit 100 Zeichen, und `anzahl` vom Typ `int` aufnehmen kann.

Nachdem Sie den Strukturtyp definiert haben, können Sie einen neuen Artikel definieren:

```
struct artikel pulver;
```

Damit haben Sie zunächst einen neuen Artikel »pulver« definiert. In C++ können Sie im Gegensatz zu C auch das Schlüsselwort `struct` weglassen:

```
// in C++ erlaubt - nicht aber in C
artikel pulver;
```

Dies ist allerdings wieder eine Stilfrage – ich persönlich bevorzuge das Schlüsselwort `struct`, durch das ich gleich weiß, um was es sich handelt.

Die obenstehende Anweisung

```
struct artikel pulver;
```

hätten Sie mit folgender Schreibweise gleich bei der Deklaration erreichen können:

```
struct artikel {
   int sachnummer;
   char bezeichnung[100];
   int anzahl;
} pulver;
```

So könnten Sie auch gleich mehrere Typen auf einmal deklarieren:

```
struct artikel {
   int sachnummer;
   char bezeichnung[100];
   int anzahl;
} pulver, stahl, rohre;
```

Hier haben Sie gleich drei erweiterte Variablen vom Typ `artikel` deklariert.

Zugriff auf die Strukturmitglieder

Um jetzt auf die einzelnen Strukturmitglieder zuzugreifen, wird der Punktoperator wie folgt verwendet:

```
VariablenBezeichner.StrukturMitglied
```

Im Beispiel des Artikels »pulver« greifen Sie auf die Strukturvariable `sachnummer` wie folgt zu:

```
pulver.sachnummer
```

Ansonsten erfolgt die weitere Behandlung und Verwendung wie bei den Basisdatentypen, was das folgende Beispiel demonstriert:

```
// struct1.cpp
#include <iostream>
#include <cstring>
using namespace std;

struct artikel {
   int sachnummer;
```

```
   char bezeichnung[100];
   int anzahl;
};

int main(void) {
   struct artikel pulver;
   pulver.sachnummer = 1234;
   strncpy (
       pulver.bezeichnung,
       "Eisenpulver 0.3mm (1kg)",
       sizeof(pulver.bezeichnung)-1 );
   pulver.anzahl = 10;

   cout << pulver.sachnummer << '\n';
   cout << pulver.bezeichnung << '\n';

   cout << pulver.anzahl << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
1234
Eisenpulver 0.3mm (1kg)
10
```

Strukturen können natürlich ebenso wie die Basisdatentypen direkt bei der Deklaration mit Werten initialisiert werden:

```
struct artikel {
   int sachnummer;
   char bezeichnung[100];
   int anzahl;
} pulver = {
   1234,
   "Eisenpulver 0.3mm (1kg)",
   10
};
```

Oder auch erst bei der Definition der Variablen, zum Beispiel in der `main()`-Funktion:

```
struct artikel pulver = {
   1234,
   "Eisenpulver 0.3mm (1kg)",
   10
};
```

Parameterübergabe mit Strukturen an Funktionen

Die Parameterübergabe von Strukturen an Funktionen lässt sich im Prinzip genauso wie mit den Basisdatentypen realisieren. Auch hier steht Ihnen die Möglichkeit zur Verfügung, die Daten per *call by value* als Kopie oder mit *call by reference* als Adresse zu übergeben.

Es wurde bereits erwähnt, dass man umfangreichere Typen nicht als Kopie an die Funktion übergeben sollte, um aufwendiges Kopieren auf dem Stack zu vermeiden. Umfangreiche Strukturen kommen zum Teil auf enorme Datenmengen. Somit empfiehlt sich also bei Funktionen das *call by reference*-Verfahren.

Wenn zum Beispiel der Prototyp

```
void print( const struct astruct *p );
```

gegeben ist, stellt sich die Frage, wie man in der Funktion die einzelnen Elemente der Struktur dereferenzieren kann. Wenn Sie Folgendes schreiben

```
*p.strukturVariable
```

würde sich der Compiler beschweren. Der Grund dafür ist, dass der Punktoperator eine höhere Bindungskraft hat als der Indirektionsoperator. Dieses Problem kann man mit einer Klammerung umgehen. Diese wiederum hat eine höhere Bindungskraft (siehe auch Anhang des Buches) als der Punktoperator.

```
(*p).strukturVariable
```

Solch eine Schreibweise ist allerdings auch nicht einfach zu lesen und leider auch sehr fehleranfällig. Daher wurde der Operator `->` – in Form eines Pfeils oder Zeigers (also recht passend) – eingeführt. Somit sieht eine Dereferenzierung hiermit wie folgt aus:

```
p->strukturVariable
```

In der Praxis sieht dies so aus:

```
// struct2.cpp
#include <iostream>
#include <cstring>
using namespace std;

struct astruct {
   int iwert;
   float fwert;
};
```

```
void print( const struct astruct *p );

int main(void) {
   struct astruct atest;
   atest.iwert = 10;
   atest.fwert = 11.11;
   print( &atest );
   return 0;
}

void print ( const struct astruct * p ) {
   cout << "iwert : " << p->iwert << '\n';
   cout << "fwert : " << p->fwert << '\n';
}
```

Das Programm bei der Ausführung:

```
iwert : 10
fwert : 11.11
```

Natürlich lässt sich das auch mit den Referenzen nachbilden. Allerdings muss man dann wieder mit dem Punktoperator auf die einzelnen Elemente zugreifen, und der Funktionsaufruf erfolgt ohne den Adressoperator:

```
...
print( atest );
...
void print ( const struct astruct& p ) {
   cout << "iwert : " << p.iwert << '\n';
   cout << "fwert : " << p.fwert << '\n';
}
```

Rückgabewert von Strukturen aus Funktionen

Wie schon bei der Parameterübergabe von Strukturen an Funktionen gilt auch bei der Rückgabe, dass *call by reference* dem *call by value* vorzuziehen ist, um den Aufwand auf dem Stack zu minimieren. Im folgenden Beispiel wird der Speicher für die Struktur erst zur Laufzeit in der Funktion angefordert, dann werden die entsprechenden Werte zugewiesen und am Ende dem Aufrufer zurückgegeben.

```
// struct3.cpp
#include <iostream>
#include <cstring>
using namespace std;

struct astruct {
   int iwert;
```

```
   float fwert;
};

void print( const struct astruct* p );
struct astruct* input( int i, float f );

int main(void) {
   struct astruct* atest;
   atest = input( 10, 11.11 );
   print( atest );
   return 0;
}

void print ( const struct astruct* p ) {
   cout << "iwert : " << p->iwert << '\n';
   cout << "fwert : " << p->fwert << '\n';
}

struct astruct* input( int i, float f ) {
   // Speicher anfordern für die Struktur
   struct astruct* p = new (struct astruct);
   // Werte zuweisen
   p->iwert = i;
   p->fwert = f;
   // Adresse zurückgeben
   return p;
}
```

Das Programm bei der Ausführung:

```
iwert : 10
fwert : 11.11
```

Vergleichen von Strukturen

Ein direkter Vergleich zweier Strukturen ist gewöhnlich nicht möglich, da der Compiler aus Optimierungsgründen die einzelnen Elemente nicht immer direkt aufeinander folgend im Speicher anordnet.

Da hilft auch kein Vergleich Byte für Byte auf niedriger Ebene, da auch hier die Ergebnisse durch zufällig gesetzte Bits in diesen »Lücken« verfälscht sein könnten. Zwar besitzen viele Compiler einen Schalter, mit dem man die Lücken entfernen kann, aber wie schon herauszuhören ist, ist dies compilerabhängig und somit nicht portabel.

Für einen direkten Vergleich von Strukturen werden Sie also nicht darum herumkommen, die einzelnen Elemente einer Struktur miteinander zu vergleichen. In der Praxis könnte ein solcher Vergleich wie folgt aussehen:

```
// struct4.cpp
#include <iostream>
using namespace std;

struct astruct {
   int iwert;
   float fwert;
};


bool cmp_struct ( const struct astruct* s1,
                  const struct astruct* s2 );

int main(void) {
   struct astruct atest1 = { 10, 11.11 };
   struct astruct atest2 = { 10, 12.11 };
   if( (cmp_struct( &atest1, &atest2 )) == true ) {
      cout << "Inhalt ist gleich\n";
   }
   else {
     cout << "Inhalt ist nicht gleich\n";
   }
   return 0;
}

bool cmp_struct ( const struct astruct* s1,
                  const struct astruct* s2 ) {
   if( (s1->iwert == s2->iwert) &&
       (s1->fwert == s2->fwert)     ) {
      return true;  // Inhalt beider Strukturen gleich
   }
   else {
      return false; // Strukturen sind nicht gleich
   }
}
```

Arrays von Strukturen

Natürlich können Sie auch ganze Arrays von Strukturen verwenden. Arrays sind schließlich nicht nur auf die Basistypen beschränkt und lassen sich auch ohne

Schwierigkeiten mit den fortgeschrittenen Typen wie den Strukturen oder Klassen kombinieren:

```
struct astruct {
   int iwert;
   float fwert;
};
```

Wollen Sie jetzt ein Array aus zehn Elementen definieren, gehen Sie folgendermaßen vor:

```
struct astruct strarray[10];
```

Hiermit hätten Sie ein Array mit zehn Elementen definiert, bei dem jedes Element aus der Struktur `astruct` besteht. Werte können Sie den einzelnen Elementen so zuweisen:

```
int i = 0;
// Elemente mit dem Index 0
strarray[i].iwert = 10;
strarray[i].fwert = 11.11;
// nächstes Element
i++;
```

Natürlich können Sie solch ein Array auch direkt bei der Deklaration mit Werten initialisieren. Dies geschieht ähnlich wie bei den mehrdimensionalen Arrays:

```
struct astruct strarray[3] = {
   { 10, 11.11 },
   { 11, 12.12 },
   { 12, 13.13 }
}
```

Das folgende Beispiel demonstriert die Verwendung der Arrays von Strukturen in der Praxis. Dabei wird eine einfache Geburtstagsverwaltung mit folgender Struktur erstellt:

```
struct geburtstage {
   char name[50];
   int tag;
   int monat;
   int jahr;
};
```

Im Programm wird ein Array deklariert, mit dem Sie zehn Geburtstagsdaten speichern können:

```
struct geburtstage geb[DATA];
```

Im Grunde ein einfaches Listing ohne besondere Extras:

```
// struct5.cpp
#include <iostream>
using namespace std;

struct geburtstage {
   char name[50];
   int tag;
   int monat;
   int jahr;
};

const int DATA = 10;

void input( int& zaehler, struct geburtstage* b );
void output( int zaehler, struct geburtstage* b );

int main(void) {
   struct geburtstage geb[DATA];
   int zaehler = 0, wahl;
   do {
      cout << "\nGeburtstage-Verwaltung\n";
      cout << "----------------------\n";
      cout << "-1- Eingeben\n";
      cout << "-2- Ausgeben\n";
      cout << "-3- Ende\n";
      cout << "Ihre Auswahl bitte : ";
      cin >> wahl;
      switch( wahl ) {
         case 1 :
            input( zaehler, geb ); break;
         case 2 :
            output( zaehler, geb ); break;
         case 3 :
            break;
         default:
            cout << "Falsche Eingaben!\n";
      }
   } while( wahl != 3 );
   return 0;
}

void input( int& zaehler, struct geburtstage* b ) {
   cout << "Name  : ";
```

```
   cin >> b[zaehler].name;
   cout << "Tag   : ";
   cin >> b[zaehler].tag;
   cout << "Monat : ";
   cin >> b[zaehler].monat;
   cout << "Jahr  : ";
   cin >> b[zaehler].jahr;
   // Wichtig!!! Index erhöhen!!!
   zaehler++;
}

void output( int zaehler, struct geburtstage* b ) {
   for( int i = 0; i < zaehler; i++ ) {
      cout << b[i].name  << " - "
           << b[i].tag   << '.'
           << b[i].monat << '.'
           << b[i].jahr  << '\n';
   }
   cout << "\n\n";
}
```

Das Programm bei der Ausführung:

```
Geburtstage-Verwaltung
----------------------
-1- Eingeben
-2- Ausgeben
-3- Ende
Ihre Auswahl bitte : 1
Name  : Zapf
Tag   : 11
Monat : 11
Jahr  : 1988

Geburtstage-Verwaltung
----------------------
-1- Eingeben
-2- Ausgeben
-3- Ende
Ihre Auswahl bitte : 2
Wolf - 12.11.1974
Zapf - 11.11.1988
```

Auch wenn sich die Verwendung von Arrays kombiniert mit Strukturen ganz toll anhören mag, so wird in der Praxis doch eher auf dynamische Datenstrukturen zurückgegriffen. Denn bei den Arrays von Strukturen ist irgendwann Schluss –

man kann zwar den Indexwert ausreichend groß dimensionieren, aber man sollte bedenken, dass dieser Speicherplatz vom Programm verwendet wird. Bei umfangreichen Strukturen mit mehreren hundert Elementen sind schnell mal ein paar Megabytes für ein einfaches Programm verbraucht.

Will man die Daten dann auch noch sortieren oder einige Daten löschen (und die Lücken füllen), wird aufwendiges Hin- und Herkopieren mit einem temporären Speicher nötig, was einen weiteren Performanceverlust zur Folge hat.

Solche Probleme können Sie zum Beispiel durch verkettete Listen minimieren. Hier wird Speicherplatz auf Anfrage vom Heap erzeugt, und die Daten werden gleich richtig einsortiert. Gelöschte Daten werden ohne großen Aufwand »ausgehängt« und zerstört.

Strukturen in Strukturen

Natürlich lassen sich auch Strukturen innerhalb von Strukturen verwenden – es wird hierbei auch von *Nested Structures* gesprochen. Dies lässt sich recht gut an unserem Beispiel der Geburtstagsverwaltung demonstrieren. Nehmen wir an, Sie haben diese Struktur ein wenig erweitert:

```
struct geburtstage {
   char name[50];
   char vname[50];
   char email[100];
   char wohnort[100];
   // ... usw.
   int tag;
   int monat;
   int jahr;
};
```

Bei umfangreichen Strukturen macht es häufig Sinn, einzelne Daten in gesonderten Strukturen abzulegen, um das Programm wieder lesbarer zu machen. Das Beispiel »Geburtstage« könnte man demnach wie folgt zerlegen (mit denselben Daten):

```
struct datum {
   int tag;
   int monat;
   int jahr;
};

struct person {
   char name[50];
```

```
   char vname[50];
   char email[100];
   char wohnort[100];
   // ... usw.
};

struct geburtstage {
   struct datum geb_datum;
   struct person geb_person;
};
```

Eine Variable können Sie jetzt folgendermaßen erzeugen:

```
struct geburtstage geb;
```

Das Ganze lässt sich natürlich auch als Array verwenden. Der Zugriff auf die einzelnen Elemente funktioniert jetzt etwas anders. Wollen Sie zum Beispiel den Namen der Struktur `person` eintragen, so müssen Sie folgendermaßen vorgehen:

```
strncpy( geb.geb_person.name, "Wolf", 50 );
```

Oder das Geburtsjahr lässt sich so zuweisen:

```
geb.geb_datum.jahr = 1974;
```

Natürlich lässt sich auch wieder eine direkte Zuweisung bei der Definition durchführen:

```
struct geburtstage geb = {
   { 12, 11, 1974 },
   { "Wolf", "Jürgen", "email@email.de", "/home/mering" }
}
```

In der Praxis wird man so etwas dynamisch mit einem Zeiger machen wollen, dafür muss man auf den Pfeiloperator zugreifen:

```
// Speicher vom Heap anfordern
struct geburtstage *geb = new (struct geburtstage);
...
strncpy( geb->geb_person.name, "Wolf", 50 );
...
geb->geb_datum.jahr = 1974;
```

Ein weiterer Vorteil der »Verfeinerung« von Strukturen besteht darin, dass Sie die einzelnen Strukturen jederzeit auch in ganz anderen Strukturen oder für sich allein verwenden können – deshalb erzeugen Sie ja einen fortgeschrittenen Typ. So können Sie jederzeit im Programm die Struktur `datum` für sich verwenden:

```
struct datum d = { 12, 11, 1974 };
```

Aber auch die mehrfache Verwendung einer Struktur in einer Struktur ist möglich und oft recht sinnvoll:

```
struct personal {
   struct datum einstellungs_datum;
   struct datum geburts_datum;
   struct person person_daten;
};
```

Auch hier gilt natürlich, wie schon am Ende von Arrays und Strukturen erwähnt, dass man in der Praxis eine verkettete Liste einsetzen sollte, wenn man mehrere Daten erfassen will. Hierzu ist eigentlich nicht mehr viel nötig – es bedarf nur noch eines Zeigers auf die Adresse einer Struktur, die demselben Typ entspricht:

```
struct personal {
   struct datum einstellungs_datum;
   struct datum geburts_datum;
   struct person person_daten;
   struct personal *next;
};
```

Hier haben Sie praktisch einen Zeiger vom Typ `struct personal` definiert, der auf ein Objekt desselben Typs zeigen kann. In der Praxis ist dies das nächste Element in der Kette. Man hängt hiermit die Daten aneinander wie bei einer Perlenkette.

Verkettete Listen werden in C++ gewöhnlich mit dem OOP-Ansatz erstellt (siehe auch Abschnitt 4.8.9, »Fallbeispiel: Verkettete Listen«). Dennoch gehören sie zu den grundlegenden Dingen, die ein Programmierer zumindest kennen sollte, weil man dadurch wirkliche Optimierungen im Hinblick auf Geschwindigkeit, Speicherverbrauch usw. erzielen kann. Aus diesem Grund finden Sie hier einen kleinen Exkurs darüber, was zu den Grundlagen einer verketteten Liste gehört.

[»] **Hinweis**

Detaillierter und umfangreicher wird auf die verketteten Listen in dem Buch »C von A bis Z« eingegangen. Sie finden dieses Buch als Openbook auf der Buch-CD. Zwar wird hier als Sprache C verwendet, aber bezüglich der Implementierung gibt es keine Unterschiede. In C werden lediglich andere Funktionen für die Speicherverwaltung (`malloc()` und Co.) und die Ein- bzw. Ausgabe verwendet.

Exkurs: Verkettete Listen

Im Gegensatz zu den Arrays gehören verkettete Listen zu den dynamischen Datenstrukturen, die die Speicherung einer unbestimmten Anzahl zusammengesetzter Datentypen erlauben. Bei den Arrays müssen die einzelnen Speicherzei-

len im Speicher immer der Reihe nach abgelegt sein. Nicht so bei den verketteten Listen, hier sind die Speicherorte absolut referenziert.

Verkettete Listen oder Arrays

Häufig wird gefragt, was man denn nun verwenden soll. Dies hängt vom Anwendungsfall ab und kann nicht pauschal beantwortet werden. Der Vorteil von verketteten Listen besteht darin, dass man sich keine Gedanken darüber zu machen braucht, wie viele Elemente maximal gespeichert werden. Man erzeugt einfach ein Element und fügt es in der Liste ein. Benötigt man es nicht mehr, kann man es wieder löschen. Die Anzahl der möglichen Elemente ist nur durch den verfügbaren Speicher beschränkt. Allerdings ist die Zugriffszeit auf die Elemente an einer bestimmten Position länger als in einem Array (sofern es sich nicht um das erste Element handelt).

Die Daten (der Knoten)

Als Basis für die verkettete Liste wird ein Knoten (engl.: node) verwendet. Dieser Knoten wird als Element in der Liste bezeichnet und enthält die Daten und einen Zeiger auf seinen Nachfolger. Ein solcher Knoten wird als einfache Struktur realisiert:

```
struct Knoten {
   int daten;
   Knoten* next;
};
```

Natürlich kann ein solcher Knoten mehr Daten, als hier mit `daten` verwendet, enthalten. Für den Anfänger ist die Verwendung des Typs `Knoten` innerhalb der Deklaration der Struktur `Knoten` ein wenig verwirrend. Hierbei wird lediglich ein Zeiger auf ein Knotenelement definiert. Und da ein Zeiger immer dieselbe Speichergröße hat (egal, von welchem Typ der Zeiger ist), kann es auch für den Compiler irrelevant sein, wie der Typ `Knoten` genau aussieht. Anhand des Bezeichners des Zeigers lässt sich schon erkennen, dass dieser für das Nachfolger-Element bestimmt ist.

Der Anfang der Liste (der Anker)

Ein weiteres Element, das man bei einer verketteten Liste benötigt, ist ein Anfang der Liste, was häufig auch als *Anker* (engl.: anchor) oder als *Start* bezeichnet wird. Über diesen Anker erfolgt der Zugriff auf das erste Element im Knoten. Nach dem Zugriff auf das erste Element der Liste werden die weiteren Elemente mit dem Zeiger `next` (ein Verweis auf den nächsten Listenknoten) erreicht.

Das Ende der Liste

Der letzte Knoten in der Liste zeigt auf einen Nullwert. Im Beispiel kann man sich so lange von einem Element zum nächsten hangeln, bis `next 0` ist. Mit `0` zeigen Sie das Ende der Liste an. Nach dem Starten des Programms ist die Liste gewöhnlich leer, daher muss zu Beginn auch der Anfang der Liste (der Anker) `0` enthalten.

Eine verkettete Liste sieht also etwa so aus, wie es in der folgenden Abbildung schematisch dargestellt ist:

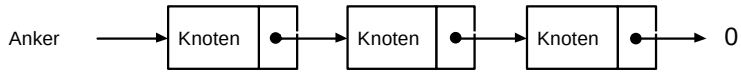

Abbildung 2.11 Schematische Darstellung einer einfach verketteten Liste

Ein einfaches Beispiel

Das folgende Listing zeigt ein einfaches Beispiel einer solchen verketteten Liste. Hierbei wurden lediglich die Funktionen zum Einfügen von Elementen am Ende, zum Anzeigen aller Elemente und zum Löschen einzelner Elemente implementiert:

```
// list.cpp
#include <iostream>
using namespace std;

struct Knoten {
   int daten;
   Knoten* next;
};

// Anfang der Liste
Knoten* Anfang = 0;

// Funktions-Prototypen
Knoten* insertKnoten( int& val);
void showKnoten( const Knoten* n );
Knoten* deleteKnoten( int dat );

int main(void) {
   Knoten* node;
   int auswahl, ival;
   do {
      cout << "Eine einfache verkettete Liste\n";
      cout << "------------------------------\n";
      cout << "-1- Neues Element hinzufügen\n";
      cout << "-2- Alle Elemente ausgeben\n";
```

```
      cout << "-3- Einzelnes Element löschen\n";
      cout << "-4- Programm beenden\n\n";
      cout << "Ihre Auswahl : ";
      cin >> auswahl;
      switch( auswahl ) {
         case 1 :
            cout << "Daten eingeben: ";
            cin >> ival;
            node = insertKnoten( ival );
            break;
         case 2:
            showKnoten( node );
            break;
         case 3:
            cout << "Wert zum Löschen eingeben: ";
            cin >> ival;
            node = deleteKnoten( ival );
            break;
         case 4:
            break;
         default:
            cout << "Falsche Menüauswahl?\n";
      }
   }  while( auswahl != 4);
   return 0;
}

// Funktion zum Einfügen neuer Elemente
Knoten* insertKnoten( int& val ) {
   // Ist noch kein Element in der Liste,
   // dann fügen wir das erste am Anfang ein
   if( Anfang == 0 ) {
      Knoten* node = new Knoten;
      node->daten = val;
      node->next = 0;
      Anfang = node;
      return Anfang;
   }
   // Es sind bereits Elemente in der Liste,
   // dann soll das neue hinten angehängt werden
   else {
      Knoten* node = Anfang;
      Knoten* newNode;
      while( node->next != 0 )
         node=node->next;
```

```
      newNode = new Knoten;
      newNode->daten = val;
      newNode->next = 0;
      node->next = newNode;
      return Anfang;
   }
}

// Alle Elemente der Liste anzeigen
void showKnoten( const Knoten* n ) {
   if ( Anfang == 0 ) {
      cout << "Die Liste ist leer\n";
   }
   else {
      cout << "1. Element: " << n->daten << '\n';
      for( int i = 2; n->next != 0; i++ ) {
         n=n->next;
         cout << i << ". Element: " << n->daten << '\n';
       }
   }
}

// Das erste Element mit dem Wert dat aus der Liste löschen
Knoten* deleteKnoten( int dat ) {
   if ( Anfang == 0 ) {
      cout << "Die Liste ist leer\n";
   }
   // Ist das erste Element das von uns gesuchte?
   if( Anfang->daten == dat ) {
      Knoten* del = Anfang;
      if( Anfang->next != 0 )
         Anfang = Anfang->next;
      delete del;
   }
   // Die komplette Liste nach dem gesuchten
   // Element durchlaufen
   else {
     Knoten* node = Anfang;
     while( node->next != 0 && node->next->daten != dat )
        node=node->next;
     if( node->next == 0 )
        cout << "Element zum Löschen kommt nicht" <<
                " in der Liste vor!\n";
```

```
      else {
         // das zu löschende Element an del zuweisen
         Knoten* del = node->next;
         // Einen Hilfszeiger hinter das zu löschende Element
         Knoten* help = del->next;
         // das zu löschende Element "aushängen"
         node->next = help;
         delete del;
       }
    }
    return Anfang;
}
```

Das Programm bei der Ausführung:

```
Eine einfache verkettete Liste
------------------------------
-1- Neues Element hinzufügen
-2- Alle Elemente ausgeben
-3- Einzelnes Element löschen
-4- Programm beenden

Ihre Auswahl :
```

Anhand des Listings lassen sich allerdings auch die Nachteile einer solchen verketteten Liste erkennen. Der Aufwand, den man betreiben muss, um nach Daten zu suchen, Knoten einzufügen oder zu löschen, die Liste zu sortieren etc., ist sehr groß, da über jedes Element gegangen werden muss. Natürlich hat man auf der anderen Seite immer den Vorteil, dass verkettete Listen einen sehr geringen Speicherbedarf haben.

Das Einfügen am Anfang der Liste hingegen geht wieder relativ schnell. Verwendet man einen weiteren Zeiger für das Ende der Liste, der immer auf das letzte Element zeigt, können die Daten auch sehr schnell am Ende der Liste angehängt werden. Natürlich benötigt die Liste dadurch wieder etwas mehr Speicherplatz.

Doppelt verkettete Listen

Bei den doppelt verketteten Listen hat jedes Element der Liste nicht nur einen Zeiger auf das noch folgende, sondern auch einen zusätzlichen zweiten Zeiger auf das Vorgänger-Element.

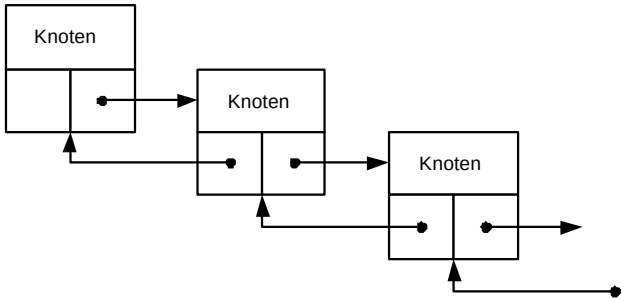

Abbildung 2.12 Schematische Darstellung einer doppelt verketteten Liste

Auf unser Beispiel bezogen, sieht die Struktur des Knotens folgendermaßen aus:

```
struct Knoten {
   int daten;
   Knoten* next;     // Zeiger auf den Nachfolger
   Knoten* previous; // Zeiger auf den Vorgänger
};
```

Gewöhnlich zeigt der Vorgänger-Zeiger des ersten Elements auf ein Dummy-Element – ebenso wie der Nachfolger-Zeiger auf das letzte Element. Diese beiden Dummy-Elemente werden zum Auffinden von Anfang und Ende der doppelt verketteten Liste verwendet.

Der Vorteil einer doppelt verketteten Liste ist, dass jetzt die Elemente auch von hinten nach vorne durchlaufen (iteriert) werden können. Ist die Liste sortiert, können die Elemente in der zweiten Hälfte noch schneller gefunden werden. Natürlich benötigt die doppelt verkettete Liste mehr Speicherplatz, da ein zusätzlicher Zeiger verwendet wird.

[»] **Hinweis**

Der OOP-Ansatz, der das Prinzip der Kapselung verwendet, wird in Abschnitt 4.8.9, »Fallbeispiel: Verkettete Listen«, beschrieben. Des Weiteren wird auch auf die von STL angebotene (objektorientierte) Schnittstelle zu den verketteten Listen eingegangen (siehe Abschnitt 5.3.5, »Container«).

Bit-Felder (gepackte Strukturen)

Bit-Felder sind nichts anderes als gepackte Strukturen. Sie sollen den Speicherplatz einer Struktur so weit wie möglich reduzieren. Dazu beispielsweise folgende Struktur:

```
struct data {
   unsigned int links;
   unsigned int rechts;
```

```
   unsigned int vor;
   unsigned int zurueck;
   unsigned int status;
};
```

Jedes einzelne Element dieser Struktur belegt auf einem 32-Bit-Rechner gewöhnlich vier Bytes. Die komplette Struktur mit allen fünf Elementen benötigt somit 20 Bytes an Speicherplatz.

Die Struktur soll einen Roboter im Raum steuern. Bei der Richtungssteuerung benötigt man lediglich die Werte `1` und `0`. `1` für »Richtung aktiv« und `0` für »nicht aktiv«. Außerdem soll noch ein Wert für den Status verwendet werden, der einen Fehlercode (sagen wir von 0–16) zurückgibt, dessen Bedeutung jetzt allerdings sekundär ist.

Hier bietet Ihnen C++ die Möglichkeit, die einzelnen Strukturelemente mit Bit-Arrays zu verwenden. Dazu müssen Sie lediglich hinter den Strukturelementen einen Doppelpunkt einfügen, gefolgt von der Anzahl der Bits, die das Element verwenden soll:

```
struct data {
   unsigned int links   : 1;
   unsigned int rechts  : 1;
   unsigned int vor     : 1;
   unsigned int zurueck : 1;
   unsigned int status  : 4;
};
```

Hiermit fordern Sie den Compiler auf, für die Richtungen `links`, `rechts`, `vor` und `zurück` jeweils ein Bit zu verwenden und für das Element `status` vier Bit. Dies wären insgesamt acht Bit und somit ein Byte.

Gewöhnlich werden für diese Struktur aber dennoch vier Bytes verwendet, da, wie bereits erwähnt, der Compiler aus Optimierungsgründen ein bestimmtes *Alignment* (bei 32-Bit-Rechnern meistens ein Vier-Byte-Alignment) verwendet. Nicht verwendeter Speicher (hier die drei Bytes) wird auch hier wieder »aufgefüllt« und bleibt ungenutzt. Es wird auch vom *Padding* (Auffüllen, Polsterung) des Speichers gesprochen.

Viele Compiler besitzen daher einen speziellen Schalter, mit dem diese Lücke entfernt werden kann. Mit dem Schalter

```
__attribut__
```

können dem Compiler mehrere Informationen zu einer Funktion, zu Variablen oder Datentypen übergeben werden. Um damit eine lückenlose Speicherbelegung zu erreichen, könnten Sie das Attribut `packed` verwenden:

```
struct data {
   unsigned int links:1;
   unsigned int rechts:1;
   unsigned int vor:1;
   unsigned int zurueck:1;
   unsigned int status:4;
}__attribute__ ((packed));
```

Sollte dieser Schalter bei Ihrem Compiler nicht funktionieren, können Sie auch das Pragma `pack` verwenden:

```
#pragma pack(n)
```

Für `n` kann hier der Wert `1`, `2`, `4`, `8` oder `16` angegeben werden. Je nachdem, welche Angabe Sie dabei machen, wird jedes Strukturelement nach dem ersten kleineren Elementtyp oder auf *n* Byte abgespeichert.

Allerdings entferne ich mich jetzt ziemlich vom Thema, da dies compiler-spezifisch ist und kein C++-Standard.

Ebenso ist nicht garantiert, dass die gepackte Struktur auch tatsächlich vom Compiler gepackt wird, wie hier beschrieben. Der C++-Standard schreibt nämlich nicht vor, wie die gepackten Strukturen gepackt werden sollen. Es kann somit auch sein, dass Ihr Compiler die gepackte Struktur genauso behandelt wie eine ungepackte und alles so belässt, wie es ist.

Außerdem sollte man nur gepackte Strukturen verwenden, wenn Speicherplatz knapp ist, da dies die Effizienz der Programmausführung beeinträchtigen kann.

[»] **Hinweis**

Gepackte Strukturen sind nur mit den Variablen `int` und `enum` erlaubt und funktionieren nicht mit anderen Basis- oder komplexeren Typen.

2.8.2 Unions

Neben den Strukturen können Sie auch mit Unions (auch *Variante* genannt) Daten unterschiedlichen Typs strukturieren. Abgesehen von einem anderen Schlüsselwort (`union`) besteht zwischen Unions und Strukturen zunächst kein allzu großer Unterschied. Auch die Zuweisung und der Zugriff erfolgen wie bei den Strukturen. Erst beim Umgang mit dem Speicherplatz wird der Unterschied deutlich, wie dieses Beispiel zeigt:

```
struct data {
   int iwert;
```

```
   float fwert;
};
```

Auf einem 32-Bit-Rechner belegt diese Struktur gewöhnlich acht Bytes – `int` (vier Bytes) + `float` (vier Bytes) = acht Bytes. Setzen Sie nun das Schlüsselwort `union` davor:

```
union data {
   int iwert;
   float fwert;
};
```

Nun belegt diese Struktur nur noch vier Bytes an Speicher. Durch das Schlüsselwort `union` wird Speicherplatz gespart, indem jetzt immer nur auf eines der Elemente in der Union zugegriffen werden kann:

```
union data test;

test.iwert = 10;
```

Hier haben Sie zum Beispiel dem Strukturelement `iwert` den Wert `10` zugewiesen. Würden Sie anschließend einen Wert von `fwert` zuweisen

```
union data test;

test.iwert = 10;
test.fwert = 11.11;   // Fehler!!!
```

so ist dies unzulässig. Zwar würde das Programm zunächst anstandslos laufen, aber wenn Sie jetzt zum Beispiel eine Wertzuweisung vornehmen würden wie

```
union data test;
int i;

test.iwert = 10;
test.fwert = 11.11;  // Fehler!!!
i = test.iwert;      // undefiniert
```

so wäre das weitere Verhalten undefiniert, das heißt, es ist nicht vorauszusagen, was passiert. Andersherum können Sie auch nicht mehr auf `iwert` zugreifen, wenn Sie `fwert` bereits einen Wert zugewiesen haben.

Eine Union lässt sich hervorragend einsetzen, um Daten zu organisieren – besonders, wenn man mehrere umfangreiche Strukturen hat, von denen man weiß (oder will), dass sie niemals gemeinsam verwendet werden. Haben Sie zum Beispiel zwei Strukturen mit Adressdaten definiert, von denen eine Struktur für den

privaten und die andere für den geschäftlichen Bereich eingesetzt wird, können Sie eine Union wie folgt verwenden:

```
union adressen {
   struct daten privat;
   struct daten beruf;
}
```

Hiermit lässt sich das Berufliche vom Privaten trennen.

Wenn Sie eine Union mit gleichen Datentypen definieren, ist bei folgender Verwendung immer standardmäßig das erste Element der Initialisierer:

```
union data {
   int a;
   int b;
   int c;
}
...
// Initialisiert das Strukturelement a
union data test = { 10 };
```

Hier wird automatisch das Strukturelement `a` initialisiert. Eine interessante Mixtur lässt sich mit den Unions und Arrays erstellen. Wenn Sie ein Array von Unions erstellen, müssen Sie keineswegs für die weiteren Elemente den Typ verwenden, den Sie für das erste Element eingesetzt haben. So können Sie für jedes Array-Element wieder ein anderes Strukturelement verwenden. Ein Beispiel dazu:

```
// union1.cpp
#include <iostream>
using namespace std;

union data {
   char  cwert;
   int   iwert;
   float fwert;
};

const int C = 0;
const int I = 1;
const int F = 2;

int main(void) {
   union data test[3];
   test[C].cwert = 'A';
   test[I].iwert = 10;
```

```
   test[F].fwert = 11.11;

   cout << test[C].cwert << '\t'
        << test[I].iwert << '\t'
        << test[F].fwert << '\n';
   return 0;
}
```

Auch hier entsteht zunächst wieder der Eindruck, dass diese Methode, Daten zu organisieren, praktisch ist. Das ist im Grunde auch zutreffend, aber in C++ verwendet man hierfür in der Praxis die Basis- und die abgeleiteten Klassen.

2.8.3 Aufzählungstypen

`enum` (Aufzählungstypen) können Sie als Alternative zu `const` verwenden. Diese Aufzählungstypen sind dazu gedacht, nur eine begrenzte Anzahl von Werten aufzunehmen. Wenn Sie zum Beispiel konstante Werte für einen Monat verwenden, können Sie dies statt mit `const`

```
const int JAN = 0;
const int FEB = 1;
const int MAR = 2 ;
...
const int DEC = 11;
...
...
int monat = FEB;
```

einfacher und kürzer mit `enum` machen:

```
enum monat {
  JAN, FEB, MAR, APR, MAI, JUN, JUL, AUG, SEP, OKT, NOV, DEC
 };
...
enum monat aktuell = FEB;
```

Somit gilt für eine `enum`-Anweisung folgende Syntax:

```
enum name { name1, name2, ... , nameN } bezeichner;
```

Die Namen zwischen den geschweiften Klammern von `enum` werden auch als *Tag* bezeichnet und gewöhnlich in Großbuchstaben geschrieben. Der Bezeichner ist auch hier wieder optional.

Das erste Tag bekommt standardmäßig immer den Wert `0` zugewiesen. Die weiteren Werte werden jeweils um eins inkrementiert. Dies kann aber auch geändert werden:

```
enum monat {
  JAN=1,FEB, MAR, APR, MAI, JUN, JUL, AUG, SEP, OKT, NOV, DEC
 }
```

Hier haben Sie das Tag `JAN` mit `1` vorbelegt. Somit erhöhen sich die weiteren Werte bis `DEC` (=`12`) um eins. Natürlich können Sie auch mittendrin ein Tag verändern:

```
enum farbe {
  WEISS, ROT, GRUEN, GELB=10, BLAU, SCHWARZ
 }
```

Angefangen wird bei der Farbe `WEISS` wie gewöhnlich mit `0`. Weiter wird inkrementiert bis `GELB` – das hier den Wert `10` erhält. Somit werden die weiteren Tags dahinter wiederum wie gewöhnlich mit eins inkrementiert, so dass `BLAU` den Wert `11` und `SCHWARZ` den Wert `12` besitzt. Natürlich können Sie auch alle Tags mit einem Wert vorbelegen.

Ein weiterer Vorteil von `enum` ist, dass diese Konstanten nach einer Definition fast genauso wie ein `int` verwendet werden können. Damit lassen sich Lesbarkeit und Übersicht des Quellcodes erheblich verbessern. Beispielsweise wurden folgende Aufzählungen definiert:

```
enum language { C, CPP, ASM, PHP };
```

Statt einer `int`-Variablen `language` wurde hier einfach ein Aufzählungstyp dafür definiert, den Sie wie folgt verwenden können:

```
switch( language lang ) {
   case C:
      // Sprache ist C - entsprechend reagieren
      break;
   case CPP:
      // Sprache ist C++ - entsprechend reagieren
      break;
   case ASM:
      // Sprache ist Assembler - entsprechend reagieren
      break;
   case PHP:
      // Sprache ist PHP - entsprechend reagieren
      break;
   default:
      // keine bekannte Sprache - entsprechend reagieren
}
```

Ich persönlich finde diese Verwendung wesentlich übersichtlicher und lesbarer als `int`-Variablen bzw. Konstanten.

2.8.4 typedef

Mit dem Schlüsselwort `typedef` kann ein neuer Bezeichner für einen Datentyp verwendet werden. Die Syntax einer einfachen Typdefinition sieht so aus:

```
typedef Typdefinition Bezeichner;
```

Damit lässt sich die Lesbarkeit eines Programms erheblich verbessern. Natürlich erzeugt man mit `typedef` keinen neuen Typ, sondern nur ein Synonym (Abkürzung) dafür.

Wenn Sie zum Beispiel nicht immer »unsigned int« im Programm eintippen wollen, brauchen Sie nur folgende Typdefinition vorzunehmen:

```
typedef unsigned int uint;
```

Jetzt können Sie im Programm statt

```
unsigned int iwert1, iwert2, iwert3;
```

Folgendes schreiben:

```
uint iwert1, iwert2, iwert3;
```

Das Schlüsselwort `typedef` wird ebenfalls dazu benutzt, sogenannte primitive Datentypen zu erzeugen. Wozu soll das gut sein? Nehmen wir als Beispiel den primitiven Datentyp `ichbinprimitiv_t`, der folgendermaßen definiert ist:

```
typedef long ichbinprimitiv_t;
```

Auf einem anderen System kann diese primitive Variable wie folgt definiert sein:

```
typedef unsigned int ichbinprimitiv_t;
```

Die primitiven Datentypen machen ein Programm somit portabler. Dadurch müssen Sie sich nicht mit den Datentypen bei Portierung auf andere Systeme auseinandersetzen. Wenn Sie ein Programm beispielsweise auf einem 32-Bit-System programmiert haben und dies anschließend auf einem 16-Bit-System getestet wird, kann die Suche nach dem Fehler einer falschen Werteausgabe frustrierend und zeitaufwendig sein.

Dieses Kapitel behandelt Themen, die zwar recht trocken und sehr theoretisch, für die Praxis aber unverzichtbar sind. In den ersten beiden Kapiteln haben Sie viel über verschiedene Datentypen (Basistypen, Zeiger, Referenzen, strukturierte Typen etc.) und Funktionen erfahren. In diesem Zusammenhang wurden aber noch einige wichtige Aspekte vernachlässigt, und zwar die Gültigkeitsbereiche, die Namensräume und die Sichtbarkeit. Auch die Deklarationen wurden nur recht einfach erklärt. Hierbei gibt es aber noch besondere Speicherklassenattribute, Typqualifikatoren und Funktionsattribute. Und dann fehlen auch noch die Typumwandlungen, bei denen man zwischen einer Standard-Umwandlung und einer expliziten Umwandlung unterscheidet.

3 Gültigkeitsbereiche, spezielle Deklarationen und Typumwandlungen

3.1 Gültigkeitsbereiche (Scope)

Zunächst sollte man nicht den Fehler machen, den Begriff *Gültigkeitsbereich* mit dem Begriff *Sichtbarkeit* gleichzusetzen. Ein Speicherobjekt ist sichtbar, wenn man innerhalb eines Bereichs darauf zugreifen kann. Damit ein solches Objekt sichtbar ist, muss es gültig sein. Sie können zum Beispiel ein Objekt »ungültig« machen, indem Sie es mit einem Objekt mit gleichem Namen überdecken:

```
// cu.cpp
#include <iostream>
using namespace std;

int main(void) {
   // iwert ist hier "sichtbar"
   int iwert = 100;
   {
      // iwert überdeckt das äußere iwert
      int iwert = 200;
      cout << iwert << '\n';

   }  // hier ist inneres iwert nicht mehr gültig
   // und somit ist äußeres iwert wieder sichtbar
```

```
   cout << iwert << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
200
100
```

Jetzt aber zum eigentlichen Thema, den Gültigkeitsbereichen in C++. Der Gültigkeitsbereich ist ein Abschnitt im Quellcode, in dem ein deklarierter Bezeichner verwendet werden kann. Häufig wird auch von einem *Scope* (= Gültigkeitsbereich des Bezeichners) gesprochen.

3.1.1 Lokaler Gültigkeitsbereich (Local Scope)

Wenn Sie einen Anweisungsblock verwenden, haben Sie automatisch einen lokalen Gültigkeitsbereich (*Local Scope*) eingeführt. In einem lokalen Gültigkeitsbereich gelten die dort deklarierten Bezeichner immer nur im Anweisungsblock (zwischen den geschweiften Klammern). Hierzu gehören alle Arten von Kontrollstrukturen wie `if`, `for`, `switch` und `while`, die auch als *Substatement Scope* bezeichnet werden. Wenn mehrere gleichnamige Variablen vorhanden sind, erhält immer die lokalste Variable den Zuschlag. Eine Ausnahme bildet die Verwendung des Scope-Operators (oder auch Bereichsoperator) `::` – aber dazu in Kürze mehr.

```
{
   int iwert = 100;
   {
      // Verwendet äußeres iwert
      cout << iwert << '\n';
      int iwert = 200;
      // Verwendet inneres iwert
      cout << iwert << '\n';
   }
}
```

3.1.2 Gültigkeitsbereich Funktionen

Der Gültigkeitsbereich eines Funktions-Prototyps gilt bis zum Ende der Deklaration (mitsamt Funktionsparameter), und der Gültigkeitsbereich der Funktionsdefinition erstreckt sich über die gesamte Funktion (also den Anweisungsblock der Funktion). Ansonsten gilt auch hier, dass bei der Verwendung einer globalen und einer lokalen Variable mit demselben Namen die lokalste Variable den Zuschlag erhält. Eine Ausnahme gibt es jedoch wieder, wenn Sie den Scope-Operator `::` verwenden.

Mit dem Scope-Operator haben Sie die Möglichkeit, auf den globalen Gültigkeitsbereich zuzugreifen (egal, von wo). Haben Sie zum Beispiel eine lokale und eine globale Variable mit dem Namen `iwert`, können Sie mit `::iwert` auf die globale und, wie immer, mit dem Namen `iwert` (ohne den Scope-Operator) auf die lokale Variable zugreifen. Dieser Operator wird bei den Klassen in Kapitel 4, »Objektorientierte Programmierung«, noch näher besprochen. Trotzdem hierzu ein Beispiel:

```
// scope.cpp
#include <iostream>
using namespace std;

// globale Variable
int iwert = 11;

void func( void );

int main(void) {
   func();
   return 0;
}

void func( void ) {
   int iwert = 22;
   // iwert der Funktion erhält Zuschlag
   cout << iwert << '\n';
   {
      // Nochmals iwert der Funktion
      cout << iwert << '\n';
      int iwert = 33;
      // Inneres iwert erhält Zuschlag
      cout << iwert << '\n';
      // Globales iwert erhält den Zuschlag
      cout << ::iwert << '\n';
   }
   // nochmals globales iwert
   cout << ::iwert << '\n';
}
```

Das Programm bei der Ausführung:

```
22
22
33
11
11
```

3.1.3 Gültigkeitsbereich Namensraum (Namespaces)

Ein Bezeichner, der in einem Namensraum deklariert ist, existiert vom Anfang seiner Deklaration bis zum Ende des Namensraums. Wegen der Wichtigkeit der Namensräume (engl.: *Namespaces*) in C++ wird in Abschnitt 3.2, »Namensräume (Namespaces)«, gesondert darauf eingegangen.

3.1.4 Gültigkeitsbereich Klassen (Class Scope)

Klassenelemente, die in einer Klasse deklariert werden, gelten von dieser Deklaration an bis zum Ende der Klassendeklaration. Auf Klassenelemente kann nur in Verbindung mit einer Variablen des Klassentyps zugegriffen werden. Darauf wird speziell bei den Klassen in Kapitel 4, »Objektorientierte Programmierung«, eingegangen.

3.2 Namensräume (Namespaces)

Mit Namensräumen (Namespaces) können Sie einen Gültigkeitsbereich erzeugen, in dem Sie beliebige Bezeichner wie Klassen, Funktionen, Variablen, Typen und sogar andere Namensräume deklarieren können. Die Verwendung von Namensräumen ist besonders hilfreich bei sehr umfangreichen Projekten, bei denen für gewöhnlich mit mehreren Modulen und Klassenbibliotheken gearbeitet wird. Hier kann es passieren, dass es bei der Vergabe von Namen zu Konflikten mit gleichen Namen kommt. Zwei Klassen oder Funktionen mit demselben Namen lassen sich zum Beispiel ohne Namensräume nicht verwenden.

3.2.1 Neuen Namensbereich erzeugen (Definition)

Einen neuen Namensbereich können Sie entweder global oder innerhalb eines bereits definierten Namensbereichs einführen. Dieser wird deklariert, indem hinter dem Schlüsselwort `namespace` der Name des Namensraums folgt. Die Deklarationen, die anschließend zu diesem Namensbereich gehören, werden zwischen geschweiften Klammern angegeben:

```
namespace meinBereich {
   int iwert;
   float fwert;
   // Definition von funktion()
   void funktion ( void ) {
      // Anweisungen von funktion()
   }
   // Definition von eineKlasse
```

```
    class eineKlasse {
        // Anweisungen für eineKlasse
    }
}
```

Hier haben Sie eine `int`-Variable `iwert`, eine `float`-Variable `fwert`, die Funktion `funktion` und die Klasse `eineKlasse` als Elemente für den Namensbereich `meinBereich` deklariert.

Natürlich können Sie einen solchen Namensbereich jederzeit um weitere Elemente erweitern. Im Beispiel wurden die Funktion `funktion` und die Klasse `eineKlasse` bereits definiert. In der Praxis wird allerdings meistens eine Deklaration im Namensbereich vorgenommen. Die eigentliche Definition wird für gewöhnlich außerhalb des Namensbereichs durchgeführt. Hierzu muss allerdings der Namensbereich mit dem Scope-Operator (bzw. Bereichsoperator) verwendet werden. Die Funktion `funktion` und die Klasse `eineKlasse` werden dann wie folgt definiert:

```
namespace meinBereich {
   int iwert;
   float fwert;
   // Deklaration von funktion()
   void funktion ( void );
   // Deklaration von eineKlasse
   class eineKlasse;
}

// Definition von funktion()
void meinBereich::funktion( void ) {
   // Anweisungen von funktion()
}

// Definition von eineKlasse
class meinBereich::eineKlasse {
   // Anweisungen für eineKlasse
}
```

Einen bereits eingeführten Namensbereich können Sie außerdem jederzeit wieder öffnen und um weitere Elemente erweitern oder die Definition darin vornehmen. Beispielsweise:

```
namespace meinBereich {
   int iwert;
   float fwert;
   // Deklaration von funktion()
   void funktion ( void );
```

```
   // Deklaration von eineKlasse
   class eineKlasse;
}

// Erneut im Namensbereich meinBereich
namespace meinBereich {
   // Definition von meinBereich::funktion( void )
   void funktion( void ) {
      // Anweisungen von funktion()
   }
}

// Definition von eineKlasse
class meinBereich::eineKlasse {
   // Anweisungen für eineKlasse
}

// Namensbereich neu öffnen und erweitern
namespace meinBereich {
   // Namensbereich meinBereich um dwert erweitert
   double dwert;
}
```

Jetzt könnten Sie theoretisch einen neuen Namensbereich mit denselben Funktionen erzeugen, ohne dass es hierbei zu Konflikten kommt:

```
namespace meinBereich {
   int iwert;
   float fwert;
   // Deklaration von funktion()
   void funktion ( void );
   // Deklaration von eineKlasse
   class eineKlasse;
}

namespace mein_NEUER_Bereich {
   int iwert;
   float fwert;
   // Deklaration von funktion()
   void funktion ( void );
   // Deklaration von eineKlasse
   class eineKlasse;
}
```

Hier haben Sie zwei Namensbereiche (`meinBereich` und `mein_NEUER_Bereich`) mit denselben Elementen. Das kommt zwar in der Praxis selten in dieser Form vor, soll aber hier demonstrieren, wozu Namensbereiche dienen.

3.2.2 Zugriff auf die Bezeichner im Namensraum

Verwenden Sie einen Bezeichner innerhalb eines Namensraums, so können Sie diesen, wie gewöhnlich, über den Typ und den Bezeichner ansprechen.

```
Bezeichner;
```

Wollen Sie allerdings einen Bezeichner außerhalb des Namensbereichs verwenden, müssen Sie den Namensbereich mit dem Scope-Operator mit angeben.

```
Namensbereich::Bezeichner;
```

Auf globale Bezeichner, die Sie außerhalb eines Namensbereichs definiert haben, können Sie mit dem Scope-Operator (Bereichsoperator) allein zugreifen.

```
// Aufruf des globalen Bezeichners
::Bezeichner;
```

Hierzu ein Beispiel, das die Zugriffe auf die einzelnen Bezeichner eines Namensraums demonstrieren soll. Im Beispiel wurde dreimal eine Funktion `funktion()` und dreimal der Wert `iwert` verwendet, ohne dass es zu Konflikten kommt.

```
// namespace1.cpp
#include <iostream>
using namespace std;

// Deklaration globale Funktion
void funktion( void );
// globale Variable iwert
int iwert = 1;

namespace ersterBereich {
   int iwert = 11;
   // Deklaration von Funktion im
   // Namensbereich ersterBereich
   void funktion( void );
}

namespace zweiterBereich {
   int iwert = 22;
   // Deklaration von Funktion im
   // Namensbereich zweiterBereich
   void funktion( void );
}

int main(void) {
   int gesamt;
   // Aufruf von funktion() aus Namensbereich ersterBereich
```

```
   ersterBereich::funktion();
   // Aufruf von funktion() aus Namensbereich zweiterBereich
   zweiterBereich::funktion();
   // Aufruf der globalen Funktion funktion() ...
   funktion();
   // ... oder auch
   ::funktion();

   // Zugriff auf den Wert iwert aus ersterBereich
   ersterBereich::iwert = 66;
   // Nochmals die Funktion aus dem
   // Namensbereich ersterBereich
   ersterBereich::funktion();

   // Rechnung mit Werten aus mehreren Namensbereichen
   gesamt = ersterBereich::iwert +
            zweiterBereich::iwert + ::iwert;
   cout << "Summe aus allen Namensbereichen : "
        << gesamt << '\n';
   return 0;
}

// globale Funktion definieren
void funktion( void ) {
   cout << "funktion(): Globale\n";
   cout << "Wert von iwert : " << iwert << "\n\n";
}

void ersterBereich::funktion( void ) {
   cout << "funktion(): Namensbereich ersterBereich\n";
   cout << "Wert von iwert : " << iwert << "\n\n";
}

void zweiterBereich::funktion( void ) {
   cout << "funktion(): Namensbereich zweiterBereich\n";
   cout << "Wert von iwert : " << iwert << "\n\n";
}
```

Das Programm bei der Ausführung:

```
funktion(): Namensbereich ersterBereich
Wert von iwert : 11

funktion(): Namensbereich zweiterBereich
Wert von iwert : 22
```

```
funktion(): Globale
Wert von iwert : 1

funktion(): Globale
Wert von iwert : 1

funktion(): Namensbereich ersterBereich
Wert von iwert : 66

Summe aus allen Namensbereichen : 89
```

Die Verwendung der globalen Funktion aus der `main`-Funktion heraus kann auch ohne den Scope-Operator geschehen. Wollen Sie aber zum Beispiel die globale Funktion `funktion()` aus einem anderen Namensbereich aufrufen, benötigen Sie wieder den Scope-Operator. Wenn aus dem Namensbereich `ersterBereich` die globale Funktion `funktion()` verwendet werden soll, müssen Sie wie folgt vorgehen:

```
void ersterBereich::funktion( void ) {
   cout << "funktion(): Namensbereich ersterBereich\n";
   cout << "Wert von iwert : " << iwert << "\n\n";
   // Ruft die globale Funktion funktion() auf
   ::funktion();
}
```

Den Scope-Operator zu vergessen hätte in diesem Beispiel außerdem noch den negativen Effekt, dass eine nicht mehr ohne Gewalt zu beendende Rekursion gestartet würde.

In der Praxis sollte man es sich angewöhnen, den Scope-Operator zu verwenden. Werden (wie im Beispiel gesehen) mehrere Funktionen mit demselben Namen deklariert sowie definiert, und Sie importieren einzelne oder alle Bezeichner aus einem Namensbereich, so wird dem Bezeichner aus dem importierten Bereich der Vorzug gegeben, da dieser dann eben der globalere Bezeichner ist (siehe dazu auch den folgenden Abschnitt 3.2.3.).

3.2.3 using – einzelne Bezeichner aus einem Namensraum importieren

Will man nicht ständig den Namensbereich mit angeben, kann man auch einzelne Bezeichner aus einem Namensbereich »importieren«. Hierzu werden Deklarationen mit dem Schlüsselwort `using` gemacht. Mit folgender `using`-Deklaration können Sie zum Beispiel aus dem Namensbereich `ersterBereich` die Funktion `funktion()` importieren:

```
using ersterBereich::funktion; // Achtung, ohne Klammern ()
```

Jetzt können Sie die Funktion `funktion()` aus dem Namensbereich `ersterBereich` mit einem einfachen Aufruf des Bezeichners aufrufen:

```
funktion();  // Funktion aus Namensbereich ersterBereich
```

Hierzu nochmals unser Beispiel *namespace1.cpp*, verändert durch das Schlüsselwort `using` und einige importierte Bezeichner:

```
// namespace2.cpp
#include <iostream>
using namespace std;

// Deklaration globale Funktion
void funktion( void );
// globale Variable iwert
int iwert = 1;

namespace ersterBereich {
   int iwert = 11;
   // Deklaration von Funktion
   // im Namensbereich ersterBereich
   void funktion( void );
}

namespace zweiterBereich {
   int iwert = 22;
   // Deklaration von Funktion
   // im Namensbereich zweiterBereich
   void funktion( void );
}

int main(void) {
   using ersterBereich::funktion;
   using zweiterBereich::iwert;
   int gesamt;
   // Aufruf von funktion() aus Namensbereich ersterBereich
   funktion();
   // Aufruf von funktion() aus Namensbereich zweiterBereich
   zweiterBereich::funktion();
   // Aufruf der globalen Funktion funktion() ...
   // Jetzt unbedingt mit dem Scope-Operator ::
   ::funktion();
```

```
   // Zugriff auf den Wert iwert aus zweiterBereich
   iwert = 66;
   // Nochmals die Funktion aus dem
   // Namensbereich zweiterBereich
   zweiterBereich::funktion();

   // Rechnung mit Werten aus mehreren Namensbereichen
   gesamt = ersterBereich::iwert + iwert + ::iwert;
   cout << "Summe aus allen Namensbereichen : "
        << gesamt << '\n';
   return 0;
}

// globale Funktion definieren
void funktion( void ) {
   cout << "funktion(): Globale\n";
   cout << "Wert von iwert : " << iwert << "\n\n";
}

void ersterBereich::funktion( void ) {
   cout << "funktion(): Namensbereich ersterBereich\n";
   cout << "Wert von iwert : " << iwert << "\n\n";
}

void zweiterBereich::funktion( void ) {
   cout << "funktion(): Namensbereich zweiterBereich\n";
   cout << "Wert von iwert : " << iwert << "\n\n";
}
```

Das Programm bei der Ausführung:

```
funktion(): Namensbereich ersterBereich
Wert von iwert : 11

funktion(): Namensbereich zweiterBereich
Wert von iwert : 22

funktion(): Namensbereich ersterBereich
Wert von iwert : 11

funktion(): Namensbereich zweiterBereich
Wert von iwert : 66

Summe aus allen Namensbereichen : 78
```

[»] **Hinweis**

Eine `using`-Deklaration ist die Deklaration eines Namens im Gültigkeitsbereich und keine Definition!

3.2.4 using – alle Bezeichner aus einem Namensraum importieren

Wollen Sie alle Bezeichner eines Namensraums sichtbar machen, so ist dies zum Beispiel mit folgender Angabe möglich:

```
using namespace ersterBereich;
```

Jetzt sind alle Bezeichner aus dem Namensbereich `ersterBereich` einfach mit dem Bezeichner ansprechbar – es ist keine explizite Angabe des Scope-Operators mehr nötig. Natürlich kann es trotzdem sinnvoll sein, den Scope-Operator zu verwenden, um Mehrdeutigkeiten zu vermeiden.

Beachten Sie bitte, dass eine `using`-Direktive, die den kompletten Namensbereich »importiert«, nicht einer Deklaration aller Namen eines Namensbereichs entspricht, sondern lediglich einer Bekanntgabe des Namensraums. Dies bedeutet, dass eine `using`-Direktive allein noch nicht zu einem Konflikt führt, wenn sich im aktuellen und importierten Namensbereich gleiche Bezeichner befinden. Das Problem der Mehrdeutigkeit tritt erst dann auf, wenn die Bezeichner angesprochen werden. Dazu sollte man wissen, wie C++ nach einem Bezeichner sucht (siehe Abschnitt 3.2.5, »Namensauflösung«).

[»] **Hinweis**

Beim »Importieren« einzelner Bezeichner aus einem Namensraum spricht man von einer `using`-Deklaration. Bei der Verwendung aller Namen aus dem Namensraum ohne Angabe des Scope-Operators wird von einer `using`-Direktive gesprochen.

Ein Beispiel, das die Verwendung von `using`-Direktiven demonstriert:

```
// namespace3.cpp
#include <iostream>
using namespace std;

namespace ersterBereich {
   int iwert = 11;
   // Deklaration von Funktion funktion1
   // im Namensbereich ersterBereich
   void funktion1( void );
}

namespace zweiterBereich {
```

```
   float fwert = 11.11;
   // Deklaration von Funktion funktion2
   // im Namensbereich zweiterBereich
   void funktion2( void );
}

int main(void) {
   using namespace ersterBereich;
   using namespace zweiterBereich;

   // Ruft funktion1() aus Namensbereich ersterBereich auf
   funktion1();
   // Ruft funktion2() aus Namensbereich zweiterBereich auf
   funktion2();

   // Gibt den Wert von iwert aus Bereich ersterBereich aus
   cout << "main() -> iwert : " << iwert << '\n';
   // Gibt den Wert von fwert aus Bereich zweiterBereich aus
   cout << "main() -> fwert : " << fwert << '\n';
   return 0;
}

void ersterBereich::funktion1( void ) {
   cout << "funktion(): Namensbereich ersterBereich\n";
   cout << "Wert von iwert : " << iwert << "\n\n";
}

void zweiterBereich::funktion2( void ) {
   cout << "funktion(): Namensbereich zweiterBereich\n";
   cout << "Wert von fwert : " << fwert << "\n\n";
}
```

Das Programm bei der Ausführung:

```
funktion(): Namensbereich ersterBereich
Wert von iwert : 11

funktion(): Namensbereich zweiterBereich
Wert von fwert : 11.11

main() -> iwert : 11
main() -> fwert : 11.11
```

Wenn ein Namensraum auch eine `using`-Direktive enthält, die Bezeichner importiert, so wird automatisch der zweite Namensbereich mit importiert:

```
// namespace4.cpp
#include <iostream>
using namespace std;

namespace ersterBereich {
   int iwert = 11;
   // Deklaration von Funktion funktion1
   // im Namensbereich ersterBereich
   void funktion1( void );
}

namespace zweiterBereich {
   using namespace ersterBereich;
   float fwert = 11.11;
   // Deklaration von Funktion funktion2
   // im Namensbereich zweiterBereich
   void funktion2( void );
}

int main(void) {
   // Importiert automatisch auch ersterBereich
   using namespace zweiterBereich;

   // Ruft funktion1() aus Namensbereich ersterBereich auf
   funktion1();
   // Ruft funktion2() aus Namensbereich zweiterBereich auf
   funktion2();

   // Gibt den Wert von iwert aus Bereich ersterBereich aus
   cout << "main() -> iwert : " << iwert << '\n';
   // Gibt den Wert von fwert aus Bereich zweiterBereich aus
   cout << "main() -> fwert : " << fwert << '\n';
   return 0;
}

void ersterBereich::funktion1( void ) {
   cout << "funktion(): Namensbereich ersterBereich\n";
   cout << "Wert von iwert : " << iwert << "\n\n";
}
void zweiterBereich::funktion2( void ) {
   cout << "funktion(): Namensbereich zweiterBereich\n";
   cout << "Wert von fwert : " << fwert << "\n\n";
}
```

Sie können die Namensbereiche aber auch verschachteln, das heißt, in einem Namensbereich können noch mehrere andere Namensbereiche deklariert werden. Hier ein solches Beispiel:

```
namespace ersterBereich_aussen {
   int iwert = 11;
   void funktion( void );
   namespace ersterBereich_innen1 {
      int iwert = 22;
      void funktion1( void );
   }
   namespace ersterBereich_innen2 {
      int iwert = 33;
      void funktion2( void );
   }
}
...
// Aufruf von funktion1()
ersterBereich_aussen::ersterBereich_innen1::funktion1();
// Aufruf von funktion2()
ersterBereich_aussen::ersterBereich_innen2::funktion2();
// Aufruf von funktion()
ersterBereich_aussen::funktion();
```

Wenn Sie Namensbereiche verschachteln, kann dies sehr verwirren, daher sollten Sie sich folgende Zugriffsregeln merken:

- Ein Bezeichner im inneren Namensraum verdeckt Bezeichner gleichen Namens aus den äußeren Bereichen.
- Im inneren Namensraum können Sie die Bezeichner aus den äußeren Namensbreichen verwenden, ohne den Scope-Operator dafür zu benutzen.
- Alle Bezeichner eines inneren Namensraums sind für den äußeren Namensbereich nicht sichtbar und müssen mit dem Scope-Operator verwendet werden.
- Bei verschachtelten Namensräumen, die mit der `using`-Direktive »importiert« werden, sind nur die Bezeichner im Namensraum selbst sichtbar! Bezeichner innerhalb eines Namensraums sind hier noch nicht verfügbar.

3.2.5 Namensauflösung

Hier nun der versprochene Abschnitt, der kurz beschreibt, wie C++ nach einem Bezeichner sucht (wird auch als *Name Lookup* bezeichnet). Es folgt ein Beispiel, mit dem Sie die anschließend beschriebenen Namensauflösungen in der Praxis demonstrieren können, indem Sie die einzelnen Bezeichner `iwert` auskommentieren bzw. wieder anwenden.

```
// namespace5.cpp
#include <iostream>
using namespace std;

namespace einBereich {
   // Deklaration im Namensbereich einBereich
   int iwert = 11;
}

// globale Deklaration
int iwert = 22;

int main(void) {
   using namespace einBereich;
   // lokale Deklaration
   int iwert = 33;
   cout << iwert << '\n';
   return 0;
}
```

Stößt der Compiler auf einen Bezeichner, versucht er, diesen Namen nach folgendem Schema aufzulösen:

- Zuerst wird nach der lokalen Deklaration eines Bezeichners gesucht.
- Wird keine lokale Deklaration des Bezeichners gefunden, werden Bereiche abgesucht, die im aktuellen Gültigkeitsbereich enthalten sind. Dies umfasst theoretisch auch den globalen Bereich.
- Als Letztes wird nach dem Bezeichner im importieren Namensraum gesucht. Dieser enthält auch die namenlosen Namensbereiche – siehe Abschnitt 3.2.7, »Anonyme (namenlose) Namensbereiche«.

3.2.6 Alias-Namen für Namensbereiche

Wenn ein Namensbereich bereits definiert wurde, dürfen Sie dafür auch einen neuen Namen (genauer: Alias-Namen) verwenden.

```
namespace neuerName alterName;
```

Hiermit können Sie auf die Bezeichner des Namensbereichs sowohl mit `neuerName` als auch mit `alterName` zugreifen. Diese Technik wird gerne beim Zugriff auf verschachtelte Namensbereiche verwendet, beispielsweise so:

```
namespace ersterBereich_aussen {
   int iwert = 11;
   void funktion( void );
```

```
   namespace ersterBereich_innen {
      float fwert = 11.11;
      void funktion1( void );
   }
}
```

Wenn Sie nun folgende Definition machen

```
namespace einBereich ersterBereich::ersterBereich_innen;
```

können Sie zum Beispiel mit

```
einBereich::fwert = 22.22;
```

auf die Variable `fwert` im inneren Bereich des Namensbereichs `ersterBereich_innen` zugreifen. Ohne den Alias-Namen müssten Sie umständlich Folgendes verwenden:

```
ersterBereich_aussen::erster_Bereich_innen::fwert = 22.22;
```

3.2.7 Anonyme (namenlose) Namensbereiche

Es kann auch ein Namensraum ohne einen Namen verwendet werden, beispielsweise so:

```
namespace {
   // ...
}
```

Dieser Namensbereich wird als Namensraum mit einem systemweit eindeutigen Namen mit einer `using`-Direktive verwendet und entspricht im Prinzip folgendem Konstrukt:

```
namespace einBereich {
   // ...
}
using namespace einBereich;
```

Auch wenn Sie dem anonymen Namensbereich keinen Namen gegeben haben, so wird dennoch einer vom Compiler vergeben. Durch die automatische Verwendung der anschließenden `using`-Direktive können Sie Bezeichner eines anonymen Namensbereichs im aktuellen Gültigkeitsbereich – siehe Abschnitt 3.1, »Gültigkeitsbereiche (Scope)« – direkt (ohne Scope-Operator) ansprechen.

Außerhalb dieses Gültigkeitsbereichs können die Bezeichner des anonymen Namenbereichs nicht mehr verwendet werden, weil es nicht möglich ist, eine `using`-Direktive für einen Namensbereich zu verwenden, der keinen Namen hat.

Damit hat man ein effektives Mittel, wenn man globale Variablen verwenden will/muss (warum auch immer). Da der Gültigkeitsbereich nur für die aktuelle Übersetzungseinheit (beispielsweise der Datei) gültig ist, entstehen keine Konflikte, wenn in einer anderen Datei der Übersetzungseinheit eine globale Variable mit demselben Namen existieren sollte. Natürlich stellt dies nach wie vor keinen Freischein für globale Variablen dar.

Nebenbei stellen anonyme Namensbereiche eine gute Alternative für `static`-Deklarationen da, beispielsweise:

```
static int iwert;
```

Hier haben Sie eine Variable vom Typ `int` mit einer statischen (siehe Abschnitt 3.4.3, »Speicherklasse 'static'«) Lebensdauer deklariert. Auch diese Deklaration hat ihren Gültigkeitsbereich in der aktuellen Übersetzungseinheit. Dasselbe realisieren Sie auch mit einem anonymen Namensraum wie folgt:

```
namespace {
   int iwert;
}
```

3.2.8 Namensbereich und Header-Dateien

Das beste Beispiel zum Thema Namensbereich finden Sie in der C++-Standardbibliothek selbst. Im ANSI-C++-Standard sind alle Klassen, Objekte und Funktionen in der C++-Standardbibliothek im Namensbereich `std` definiert. Durch die Verwendung von

```
using namespace std;
```

im Programm ersparen Sie sich den Scope-Operator und können auf alle Klassen, Objekte und Funktionen ohne den Scope-Operator `::` auf die Bezeichner zugreifen. Ohne die Verwendung der `using`-Direktive und das »Importieren« des Namensbereichs `std` müssten Sie auf die Bezeichner der Header-Datei `<iostream>` wie folgt zugreifen:

```
// namespace6.cpp
#include <iostream>

int main(void) {
   std::cout << "Ohne using namespace std\n";
   return 0;
}
```

Ich könnte es jetzt hierbei belassen und davon ausgehen, dass schon alles in Ordnung gehen wird. Aber wenn Sie sich vielleicht schon einige ältere Quellcodes

angesehen haben, haben Sie vermutlich festgestellt, dass es Programme gibt, die `iostream` folgendermaßen einbinden

```
#include <iostream.h>
```

(also mit `.h`) und keine `using`-Direktive für den Namensbereich `std` verwenden. Wenn Sie also das folgende Programm übersetzen, werden Sie gewöhnlich keine Probleme bekommen:

```
// namespace7.cpp
#include <iostream.h>

int main(void) {
   cout << "Ohne using namespace std\n";
   return 0;
}
```

Dass dies hier ohne Bekanntgabe des Namensbereichs `std` funktioniert, liegt daran, dass C++ vor 1997 noch gar keine Namensbereiche kannte. Die Verwendung der Header-Dateien mit der Endung *.h* wie zum Beispiel *<iostream.h>* funktioniert nur noch aus Gründen der Kompatibilität zu älteren Programmen.

Als nun eine neudefinierte Version der Standardbibliothek mit vielen neuen Hilfsmitteln hinzugefügt wurde, war der Name der Header-Dateien ein Problem. Hätte man hierbei wieder einen Namen wie *<iostream.h>* verwendet, so hätte man Probleme mit älteren Programmen bekommen. Daher hat man kurzerhand beschlossen, das *.h* aus den Standard-Header-Dateien zu entfernen.

Somit besitzen laut Standard die Header-Dateien der Standardbibliothek jetzt keine Endung mehr. Das heißt auch, dass bei vielen Compilern nun die Standard-Header-Dateien im Allgemeinen doppelt vorhanden sind – einmal mit der Endung *.h* (veraltet) und einmal ohne (aktueller Standard) –, um die Kompatibilität zu älteren C++-Programmen zu erhalten.

Bei den neuen Header-Dateien ohne eine Endung wie *<iostream>* werden die Deklarationen im Namensbereich `std` platziert. Somit ist nach dem neuen Standard Folgendes korrekt:

```
// namespace8.cpp
#include <iostream>

int main(void) {
   std::cout << "Aktueller Standard\n";
   return 0;
}
```

Oder auch mit der `using`-Direktive (mit der Sie auch gleich die Kompatibilität zu älteren Programmen bewahren, die noch keine Namensräume kennen):

```
// namespace9.cpp
#include <iostream>
using namespace std;

int main(void) {
   cout << "Aktueller Standard\n";
   return 0;
}
```

Die Version, die Sie im Beispiel *namespace7.cpp* gesehen haben, ist somit nur noch geduldet, entspricht aber nicht mehr dem aktuellen Standard. Leider kann man nicht einfach den neuen Standard verwenden. Auf manchen Implementierungen (beispielsweise auf Uralt-Systemen) schlägt nämlich die Version ohne eine Endung der Standard-Header-Dateien bei der Übersetzung fehl. Dann bleibt einem meistens nichts anderes mehr übrig, als die alte Version mit der Endung *.h* zu verwenden.

[»] **Hinweis**

Aktuellere Compiler (beispielsweise `g++` ab 4.3.x) hingegen werden ein Beispiel wie *namespace7.cpp* nicht mehr übersetzen und korrekterweise auch bemängeln, dass `cout` nicht in diesem Gültigkeitsbereich bekannt ist.

3.3 C-Funktionen bzw. -Bibliotheken in einem C++-Programm

Es wurde zuvor bereits erwähnt, dass sich der Weg, eine Header-Datei einzubinden, mit dem neuen C++-Standard geändert hat. Dies gilt auch für die Verwendung der Standardbibliotheks-Funktionen von C. Neben der Neuerung, dass Header-Dateien nicht mehr mit der Endung *.h* (siehe Abschnitt 3.2.8, »Namensbereich und Header-Dateien«) eingebunden werden, bekommen Header-Dateien von Standard-C das Zeichen »c« vorangestellt. Damit werden diese als C-Header-Dateien gekennzeichnet und gleich sichtbar. Aus <*stdio.h*> wird also <*cstdio*>.

Hierzu eine Liste von C-Header-Dateien (siehe Tabelle 3.1), wie sie im ANSI-C++-Standard eingebunden werden sollten, und das Gegenstück dazu in ANSI C:

ANSI C	ANSI C++
<assert.h>	*<cassert>*
<ctype.h>	*<cctype>*
<errno.h>	*<cerrno>*
<float.h>	*<cfloat>*
<iso646.h>	*<ciso646>*
<limits.h>	*<climits>*
<locale.h>	*<clocale>*
<math.h>	*<cmath>*
<setjmp.h>	*<csetjmp>*
<signal.h>	*<csignal>*
<stdarg.h>	*<cstdarg>*
<stddef.h>	*<cstddef>*
<stdio.h>	*<cstdio>*
<stdlib.h>	*<cstdlib>*
<string.h>	*<cstring>*
<time.h>	*<ctime>*
<wchar.h>	*<cwchar>*
<wtype.h>	*<cwtype>*

Tabelle 3.1 Vergleich der Header-Dateien in ANSI-C und ANSI C++

3.3.1 C-Funktionen aus einer C-Bibliothek aufrufen

Dieser Abschnitt richtet sich an die schon etwas erfahreneren Programmierer, die bereits Programme in C geschrieben haben.

Wenn Sie zum Beispiel eine C-Funktion aus einer C-Bibliothek aufrufen wollen, müssen Sie nicht den C-Quellcode neu kompilieren. Wollen Sie also in einem C++-Programm eine C-Funktion aus einer C-Bibliothek aufrufen, die mit einem C-Compiler übersetzt wurde, müssen Sie dies dem C++-Compiler bekanntgeben:

```
extern "C" typ funktion( parameter );
```

Damit teilen Sie dem C++-Compiler mit, dass die Funktion `funktion()` mit einem C-Compiler übersetzt wurde. Hier sollte man vielleicht noch darauf hinweisen, dass ein C++-Compiler Funktionsaufrufe anders übersetzt, weil dieser gegenüber C einige Spracherweiterungen (beispielsweise Überladung) besitzt. Natürlich können Sie hierbei auch mehrere C-Funktionen einer Bibliothek verwenden. Hierzu müssen Sie die Funktionen nur in geschweiften Klammern zusammenfassen:

```
extern "C" {
   typ funktion1( parameter );
   typ funktion2( parameter );
   typ funktion3( parameter );
}
```

Sind diese C-Funktionen bereits in einer Header-Datei deklariert, müssen Sie nur noch diese Datei einkopieren:

```
extern "C" {
   #include "CFunktionen.h"
}
```

Wollen Sie C-Funktionen oder Header-Dateien sowohl für C- als auch C++-Programme zur Verfügung stellen (was durchaus üblich ist), müssen Sie eine bedingte Kompilierung wie folgt verwenden:

```
// Haben wir ein C++-Programm
#ifdef __cplusplus
extern "C" {  // Für das C++-Programm
#endif
   // Für C- und C++-Programme
   typ funktion1( parameter );
   typ funktion2( parameter );
   typ funktion3( parameter );
#ifdef __cplusplus
}             // Für das C++-Programm
#endif
```

Mit der vordefinierten symbolischen Konstante `__cplusplus` können Sie feststellen, ob ein C- oder C++-Compiler verwendet wird. Handelt es sich um einen C++-Compiler, wird die `extern "C"`-Deklaration mit den geschweiften Klammern verwendet.

Wenn Sie C-Funktionen in einem C++-Programm neu erstellen wollen – wie dies zum Beispiel nötig ist, wenn Sie eine C-Funktion aufrufen, die als Argument eine weitere C-Funktion erwartet (etwa bei den Standard-C-Funktionen `bsearch()` und `qsort()`) –, dann können Sie auch hier den C++-Compiler mit `extern "C"` anweisen, diese Funktion als C-Funktion zu übersetzen. Hierzu ein reines C-Programmbeispiel:

```
/* bsearch.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* Anzahl der Strings */
#define MAX 5
```

```
/* Vergleichsfunktion für zwei Strings */
int cmp_str(const void *s1, const void *s2) {
   return (strcmp(*(char **)s1, *(char **)s2));
}

int main(void) {
   char *daten[MAX], puffer[80], *ptr,
        *key_ptr, **key_ptrptr;
   int count;
   /* Wörter eingeben */
   printf("Geben Sie %d Wörter ein\n", MAX);
   for (count = 0; count < MAX; count++) {
      printf("Wort %d: ", count+1);
      fgets(puffer, 80, stdin);
      /* Speicher für das Wort Number count reservieren */
      daten[count] = (char *) malloc(strlen(puffer)+1);
      strcpy(daten[count], strtok(puffer,"\n") );
   }
   /* Die einzelnen Wörter sortieren */
   qsort(daten, MAX, sizeof(daten[0]), cmp_str);
   /* Sortierte Daten ausgeben */
   for (count = 0; count < MAX; count++)
      printf("\nWort %d: %s", count+1, daten[count]);

   /* Jetzt nach einem Wort suchen */
   printf("\n\nNach welchem Wort wollen Sie suchen: ");
   fgets(puffer, 80, stdin);
   /* Zur Suche übergeben Sie zuerst den puffer an key,
    * danach benötigen Sie einen weiteren Zeiger, der
    * auf diesen Such-Schlüssel zeigt
    */
   key_ptr = strtok(puffer, "\n");
   key_ptrptr = &key_ptr;
   /* ptr bekommt die Adresse des Suchergebnisses */
   ptr =(char *) bsearch(key_ptrptr, daten, MAX,
                         sizeof(daten[0]), cmp_str);

   if(NULL == ptr)
      printf("Kein Ergebnis für %s\n", puffer);
   else
      printf("%s wurde gefunden\n", puffer);
   return EXIT_SUCCESS;
}
```

Dies hier ist reiner C-Code. Wenn Sie nun die »dankbare« Aufgabe haben, dieses Programm in ein C++-Programm umzuschreiben, haben Sie immer noch das Problem, dass die beiden Funktionen `bsearch()` und `qsort()` C-Funktionen als Argumente erwarten. Mit dem bisherigen Wissen und `extern "C"` können Sie dies durchaus selbst realisieren, wenn Sie wollen. Hier das Beispiel umgeschrieben in einen C++-Code:

```
/* bsearch.cpp */
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
using namespace std;

// Anzahl der Strings
const int MAX=5;
const int BUF=80;

// Vergleichsfunktion für zwei Strings
extern "C" {
   int cmp_str(const void *s1, const void *s2) {
      return (strcmp(*(char **)s1, *(char **)s2));
   }
}

int main(void) {
   char *daten[MAX], puffer[BUF], *ptr,
        *key_ptr, **key_ptrptr;

   // Wörter eingeben
   cout << "Geben Sie " << MAX << " Wörter ein\n";
   for (int count = 0; count < MAX; count++) {
      cout << "Wort " << (count+1) << ": ";
      cin.getline(puffer, BUF);
      // Speicher für das Wort Number count reservieren
      daten[count] = new char [BUF+1];
      strcpy(daten[count], strtok(puffer,"\n") );
   }
   // Die einzelnen Wörter sortieren
   qsort(daten, MAX, sizeof(daten[0]), cmp_str);
   // Sortierte Daten ausgeben
   for (int count = 0; count < MAX; count++)
      cout << "\nWort " << count+1 << ": " << daten[count];
```

```
   // Jetzt nach einem Wort suchen
   cout << "\n\nNach welchem Wort suchen Sie : ";
   cin.getline(puffer, BUF);
   /* Zur Suche übergeben Sie zuerst den puffer an key,
    * danach benötigen Sie einen weiteren Zeiger, der
    * auf diesen Such-Schlüssel zeigt
    */
   key_ptr = strtok(puffer, "\n");
   key_ptrptr = &key_ptr;
   // ptr bekommt die Adresse des Suchergebnisses
   ptr =(char *) bsearch(key_ptrptr, daten, MAX,
                         sizeof(daten[0]), cmp_str);

   if(0 == ptr) {
      cout << "Kein Ergebnis für " << puffer << '\n';
   }
   else {
      cout << puffer << " wurde gefunden\n";
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Geben Sie 5 Wörter ein
Wort 1: Vogel
Wort 2: Mensch
Wort 3: Auto
Wort 4: Baum
Wort 5: Pferd

Wort 1: Auto
Wort 2: Baum
Wort 3: Mensch
Wort 4: Pferd
Wort 5: Vogel

Nach welchem Wort suchen Sie : Baum
Baum wurde gefunden
```

3.4 Speicherklassenattribute

Mit Speicherklassenattributen geben Sie die Speicherklasse eines Bezeichners an und bestimmen somit die Lebensdauer und Bindung eines Bezeichners.

3.4.1 Speicherklasse »auto«

Die Speicherklasse `auto` haben Sie bisher fast immer (unbewusst) verwendet. Durch das Voranstellen des Schlüsselworts `auto` bei der Deklaration einer Variablen wird diese Variable automatisch angelegt und am Ende des Gültigkeitsbereichs (Blocks) wieder gelöscht. Der Gültigkeitsbereich von `auto` ist somit derselbe wie bei einer lokalen Variablen – siehe Abschnitt 3.1.1, »Lokaler Gültigkeitsbereich (Local Scope)«:

```
auto int iwert = 11;
```

ist dasselbe wie

```
int iwert = 11;
```

Da also `auto` die Standard-Speicherklasse ist, kann die Angabe auch ganz entfallen. Verwenden Sie `auto` für globale Bezeichner, erhalten Sie eine Fehlermeldung, da `auto` nur für lokale Objekte gültig ist – und somit auf Funktionen überhaupt nicht anwendbar, weil diese immer global sind.

3.4.2 Speicherklasse »register«

Alles, was zu `auto` gesagt wurde, trifft auch auf die Speicherklasse `register` zu. Allerdings weisen Sie mit dem Schlüsselwort `register` den Compiler an, diese Variable möglichst lange im Prozessor-Register zu halten – wodurch ein schnellerer Zugriff möglich ist als auf dem Arbeitsspeicher. Allerdings gilt für das Schlüsselwort `register` mit Variablen dasselbe wie für `inline` bei den Funktionen – der Compiler entscheidet selbst, welche Variable er in den schnellen Prozessor-Registern ablegt. Somit kann der Compiler diese Speicherklasse auch ignorieren.

3.4.3 Speicherklasse »static«

Wenn Sie eine Variable mit der Speicherklasse `static` deklarieren, existiert diese Variable von ihrer ersten Verwendung an bis zum Ende des Programms. Bei skalaren Objekten bewirkt `static`, dass dieses Objekt automatisch mit `0` initialisiert wird. Ein Beispiel dazu:

```
/* static.cpp */
#include <iostream>
using namespace std;

void funktion( void );

int main(void) {
   for( int i=0; i < 3; i++ ) {
```

```
      funktion();
   }
   return 0;
}

void funktion( void ) {
   static int iwert;
   cout << iwert << '\n';
   iwert++;
}
```

Das Programm bei der Ausführung:

```
0
1
2
```

In der Funktion `funktion()` finden Sie eine statische Variable `iwert`, deren Wert bei einem Funktionsaufruf zunächst ausgegeben wird. Beim ersten Aufruf ist dieser Wert automatisch `0` – außer man initialisiert diesen mit einem anderen Wert. Am Ende der Funktion wird dieser Wert um eins inkrementiert. Beim Beenden der Funktion wird die statische Variable `iwert` nicht wie üblich zerstört, sondern bleibt während der Lebenszeit des Programms erhalten, was die erneuten Funktionsaufrufe auch demonstrieren.

Allerdings bleibt der Gültigkeitsbereich nach wie vor nur innerhalb der Funktion `funktion()` erhalten. Es kann also nicht außerhalb der Funktion auf diese statische Variable zugegriffen werden. Ein erneuter Funktionsaufruf von `funktion()` hat außerdem den Vorteil, dass diese Variable nicht mehr erneut angelegt werden muss.

Natürlich können Sie auch Funktionen mit der Speicherklasse `static` versehen. Dies wird zum Beispiel verwendet, damit Funktionen (und natürlich auch Variablen) eine lokale Gültigkeit haben – also ausschließlich in der Datei gültig sind. In C++ ist es allerdings mittlerweile üblich, interne Objekte nicht mehr mit der Speicherklasse `static` zu versehen, sondern es wird dazu ein anonymer Namensraum verwendet – siehe Abschnitt 3.2.7, »Anonyme (namenlose) Namensbereiche«.

3.4.4 Speicherklasse »extern«

Standardmäßig können alle global in einer Datei definierten Bezeichner in anderen Dateien benutzt werden, die nicht als `static` ausgewiesen wurden.

Somit ist also die Verwendung der Speicherklasse `extern` zunächst optional – wenn man genau ist. Mit dem Schlüsselwort `extern` teilen Sie dem Compiler mit, dass die Definition von Variablen oder Funktionen in einer anderen Datei oder einem externen Modul vorgenommen wurde. Mit `extern` deklarierte Objekte geben nur den Namen bekannt und belegen zunächst noch keinen Speicherplatz. Dies geschieht erst bei der Definition in einer anderen Datei.

Hierzu ein Beispiel:

```
// abc.cpp
#include <iostream>
using namespace std;

extern void print1( void );
extern void print2( void );
extern int MAX;

int main(void) {
   print1();
   print2();
   cout << "Wert von MAX: " << MAX << '\n';
   return 0;
}
```

Hier finden Sie die Deklaration von zwei Funktionen und einer Variablen, die als `extern` deklariert sind. Somit weiß der Compiler an dieser Stelle schon mal, dass sich die Definitionen dieser Bezeichner in einer anderen Datei befinden. Im Falle der Funktionen könnten Sie das Schlüsselwort `extern` auch weglassen. Bei der Variablen `MAX` hingegen würde dies zu einer Fehlermeldung führen, wenn in der externen Datei ebenfalls eine Variable mit demselben Namen vorhanden wäre (worauf dieses Beispiel ja auch hinaus will) – wobei wir auch gleich wieder bei der Warnung wären, möglichst auf globale Variablen zu verzichten.

Die Datei, in der sich die Funktionen und die Variable befinden, heißt in diesem Fall *xyz.cpp* und sieht so aus:

```
// xyz.cpp
#include <iostream>
using namespace std;

void print1( void );
void print2( void );
int MAX =  10;

void print1( void ) {
```

```
   cout << "Ich bin print1()\n";
}

void print2( void ) {
   cout << "Ich bin print2()\n";
}
```

Sofern Sie dieses Beispiel in der Praxis ausführen wollen, müssen Sie die Datei *xyz.cpp* beim Übersetzen mit angeben bzw. dem »Projekt« hinzufügen.

Beim Einsatz von `const`-Bezeichnern lässt sich das hier Beschriebene allerdings nicht so realisieren, da `const`-Bezeichner nur in der Datei gültig sind, in der diese definiert werden. Wollen Sie `const`-Bezeichner in verschiedenen Dateien verwenden, müssen Sie diese schon bei der Deklaration als `extern` definieren. Dies sähe, bezogen auf die Datei *abc.cpp* und die Variable `MAX`, wie folgt aus:

```
// abc.cpp
...
extern const int MAX;
...
```

Und bei der Datei *xyz.cpp*, in der `MAX` auch definiert wurde, sieht der Vorgang wie folgt aus:

```
// xyz.cpp
...
extern const int MAX = 10;
...
```

Eine weitere Anwendung der Speicherklasse `extern` haben Sie ja bereits kennengelernt, als es darum ging, C-Code in einem C++-Programm einzufügen (siehe Abschnitt 3.3, »C-Funktionen bzw. -Bibliotheken in einem C++-Programm«).

3.4.5 Speicherklasse »mutable«

Die Speicherklasse `mutable` wird nur auf Klassenelemente angewendet, doch sei hier schon mal der Vollständigkeit halber darauf hingewiesen. Mit diesem Schlüsselwort spezifizieren Sie ein Klassenelement als »manipulierbar«, auch wenn dieses Element mit (nicht explizit) `const` vereinbart wurde!

3.5 Typqualifikatoren

Es gibt zwei Typqualifikatoren, `const` und `volatile`. `const` haben Sie bereits kennengelernt (siehe Abschnitt 1.5, »Konstanten«). Die beiden Qualifikatoren

lassen sich auch miteinander verwenden, wodurch sich insgesamt vier Angaben von Typen ergeben:

```
// unqualifizierte Typangabe
usigned int reg;
// const-Objekt
const unsigned int reg;
// volatile-Objekt
volatile unsigned int reg;
// const volatile-Objekt
const volatile unsigned int reg;
```

3.5.1 Qualifizierer »const«

Wie bereits erwähnt, wurde `const` bereits in Abschnitt 1.5, »Konstanten«, beschrieben. Hiermit qualifizieren Sie einen Typ so, dass dieser nicht mehr geändert werden darf. Dies bedeutet, dass er nicht mehr auf der linken Seite vor einer Zuweisung stehen darf:

```
const int MAX = 5;
// Fehler!!!
MAX = 10;
```

3.5.2 Qualifizierer »volatile«

`volatile` gibt eine Variable an, die im Programm zum Beispiel vom Betriebssystem, von der Hardware oder einem gleichzeitig ausgeführten Thread (Parallelprozess) geändert werden kann.

Der Qualifizierer `volatile` kann nur auf Variablen angewendet werden und bewirkt, dass diese Variablen außerhalb des normalen Programmablaufs den Wert verändern können. Mit `volatile` modifizieren Sie eine Variable so, dass der Wert dieser Variablen vor jedem Zugriff neu aus dem Hauptspeicher eingelesen werden muss.

Das Hauptanwendungsgebiet hierfür ist die System-, Hardware- und Treiberprogrammierung. Beispielsweise soll in einer Schleife überprüft werden, ob ein bestimmter Zustand im Register des Prozessors vorzufinden ist:

```
while ( reg &( ZUSTAND_A|ZUSTAND_B)) {
   // Ein Hardware-Gerät wird überprüft
}
cout << "Status Gerät X  ...  [OK]\n";
```

An dieser `while`-Schleife sehen manche Compiler, dass immer dieselbe Adresse überprüft wird, und optimieren diese Schleife weg. Dies bedeutet, die Überprüfung des Gerätestatus wird nur einmal ausgeführt, weil hier die Schleife weg ist. Der Compiler kann einfach nicht wissen, wie wichtig diese Überprüfung für den weiteren Programmablauf ist. Somit gilt für Variablen, die mit `volatile` deklariert sind, dass diese ohne jede Optimierung neu aus dem Hauptspeicher geladen und neue Werte auch sofort wieder dort abgelegt werden.

3.6 Funktionsattribute

Auch für Funktionen gibt es Attribute. Mit `inline` haben Sie bereits ein Attribut kennengelernt. Neben `inline` (siehe Abschnitt 1.10.6, »Inline-Funktionen«) gibt es noch die Attribute `virtual` und `explicit`, die allerdings nur im Zusammenhang mit Klassen und Methoden verwendet werden können.

3.7 Typumwandlung

Bevor man sich mit der Typumwandlung beschäftigt, sollte man sich zunächst vor Augen halten, dass ein Typ auch nichts anderes darstellt als eine Kette einzelner Bits mit 0 und 1. Wie Sie bereits bei den Basisdatentypen erfahren haben, ist die Länge dieser Bit-Kette abhängig vom Datentyp. So hat der Datentyp `char` gewöhnlich meistens ein Byte, was meistens auch acht Bits sind (aber nicht sein müssen). Der Datentyp `int` repräsentiert auf einer 32-Bit-Maschine meistens vier Bytes und hat somit gewöhnlich 32 einzelne Bits mit 0 und 1. Beispielsweise sieht die Zahl 100 als `char`-Wert wie folgt aus:

```
01100100            // 1 Byte -> 8 Bits -> Wert = 100
```

Bei einem Typ `int` sieht diese Bit-Kette wie folgt aus (32 Bit):

```
00000000 00000000 00000000 01100100
```

Im Grunde haben Sie hier dieselbe Bit-Stellung wie im ersten Byte. Somit könnten Sie für den Wert 100 sowohl den Typ `char` also auch `int` verwenden. Nehmen wir aber mal den Wert 256 als `int`:

```
00000000 00000000 000000001 00000000
```

Dieser Wert passt nicht mehr in den Typ `char`, da hierbei das zweite Byte mit einem Bit besetzt ist.

Worauf ich hinaus will, ist, dass bei einer Typumwandlung (beispielsweise hier von `int` nach `char`) einige Stellen »verloren« gehen könnten. Häufig treten solche Probleme (auch *unsichere Konvertierung* genannt) beim Konvertieren von Gleitpunktzahlen zu ganzzahligen Werten auf.

Eine Typumwandlung kann implizit – also durch den Compiler automatisch – erfolgen oder explizit vom Programmierer mit einem Cast-Ausdruck »erzwungen« werden. Bei den Klassen können Sie außerdem noch eine solche Typumwandlung selbst definieren – worauf aber erst in Abschnitt 4.7.7, »Typumwandlung abgeleiteter Klassen«, eingegangen wird.

3.7.1 Standard-Typumwandlung

Eine implizite Standard-Typumwandlung wird vom Compiler vorgenommen, wenn der Typ eines Ausdrucks nicht mit dem Typ übereinstimmt, der erwartet wurde. Dies trifft häufig in folgenden Fällen zu:

- Funktionsaufrufe – wurde bei einem Funktionsaufruf zum Beispiel ein `double`-Wert erwartet, als Argument aber ein Typ `int` übergeben, wird das `int`-Argument in einen `double`-Wert konvertiert.
- Arithmetische Ausdrücke und Vergleiche – werden bei Berechnungen zwei verschiedene Typen verwendet, so wird immer der »kleinere« der Typen in den Typ des »größeren« konvertiert.
- Initialisierung (Zuweisung) – der Wert, der auf der rechten Seite einer Zuweisung angegeben wird, wird automatisch in den Wert konvertiert, der auf der linken Seite steht.

Integral-Promotion

Um es gleich klarzustellen, wir sprechen hier von einer automatischen Promotion, die der Compiler mit den Datentypen machen »kann«, und nicht von einer automatischen Typumwandlung. Bei einer integralen Promotion wird beispielsweise häufig ein Typ in einen anderen integralen Typ umgewandelt. Ein Beispiel dazu:

```
char cwert = 0x00;
...
// Vergleich von zwei int-Werten !!!
if ( cwert != 0x80 ) {
   // ...
}
```

Obwohl Sie hier einen Typ `char` verwenden, vergleicht das ausführende Programm zwei `int`-Werte. Dass dies so ist, liegt daran, dass vor der Ausführung einer Operation immer der Typ des Operanden angepasst wird. Diese Promotion wird immer so ausgeführt, dass der »kleinere« Typ auf den »größeren« erweitert wird – natürlich nur dann, wenn auch keine Werte verloren gehen.

Es ist also tatsächlich so, dass die Typen `char`, `unsigned char`, `signed char`, `short` und `unsigned short` vom Compiler auf den Typ `int` erweitert werden können. Kann der Wertebereich nicht von `int` abgedeckt werden, so kann der Compiler versuchen, eine Anpassung auf `unsigned int` vorzunehmen. Dasselbe wird auch mit `bool`, `enum` und `wchar_t` gemacht.

Die folgende Tabelle (3.2) zeigt, welche integralen Promotionen der Compiler vornehmen kann. Auf der linken Seite steht der integrale Typ, der automatisch in den integralen Typ der rechten Seite umgewandelt wird, wenn der Wertebereich abgedeckt wird.

Typ	Integrale Promotion zu
`char`, `signed char`, `unsigned char`	`int`
`short`, `unsigned short`	`int`
`wchar_t`	`int`, `unsigned int`, `long`, `unsigned long`
`enum`	`int`, `unsigned int`, `long`, `unsigned long`
`bool`	`int`
`class` (Bit-Felder)	`unsigned int`

Tabelle 3.2 Integrale Promotion

Gleitkomma-Promotion

Neben der integralen Promotion gibt es noch die Gleitkomma-Promotion. Hier versucht der Compiler, einen Ausdruck vom Typ `float` in einen Ausdruck vom Typ `double` zu konvertieren.

Integral-Typumwandlung

Zunächst macht man sich vielleicht um die integrale Typumwandlung recht wenig Gedanken. Sobald man aber die `int`-Werte mit und ohne `signed` und `unsigned` verwendet und vermischt, sind Fehler garantiert.

Bei einer integralen Typumwandlung werden entweder `int`-Ausdrücke in einen entsprechenden `unsigned`- oder eben einen `signed`-Typ umgewandelt. Wichtig ist es zu wissen, dass die Konvertierung eines `signed`-Typs in einen `unsigned`-Typ immer undefiniert ist. Das heißt, das Ergebnis lässt sich nicht vorhersagen, weil es vom Compiler abhängt, wie dieser eine solche Konvertierung durchführt.

Ein Beispiel hierzu:

```
signed int siwert = -100;
int iwert = 100;

// iwert wird zu unsigned int konvertiert
unsigned int uiwert1 = iwert;
// undefiniert - compilerabhängig
unsigned int uiwert2 = siwert;
// Ok, iwert speichert signed int-Wert
int iwert2 = siwert;
// undefiniert - compilerabhängig
int iwert3 = 4294967290;
// Ok
unsigned int uiwert3 = 4294967290;
```

Wenn der Zieltyp `bool` ist, wird entsprechend nach `0` oder `1` konvertiert:

```
int iwert1 = 100;
int iwert2 = 0;

// int->bool - Wird nach 1 konvertiert
bool bwert1 = iwert1;
// int->bool Wird nach 0 konvertiert
bool bwert2 = iwert2;
```

Gleitkomma-Typumwandlung

Natürlich kann auch der Typ einer Gleitkommazahl in einen anderen Gleitkommatyp umgewandelt werden (beispielsweise `float` nach `double` oder umgekehrt). Sofern allerdings der Wert einer Gleitpunktzahl nicht in einer anderen ganz abgedeckt werden kann, ist der Wert auf der linken Seite der Zuweisung undefiniert – also auch wieder compilerabhängig, da sich nicht sagen lässt, was der Compiler damit macht.

Integral-Gleitkomma-Typumwandlung

Natürlich kann auch einem integralen Typ ein Gleitkommatyp zugewiesen werden. Allerdings kann der integrale Typ keine Nachkommastelle speichern, so dass diese Informationen verloren gehen. Lässt sich der Gleitkommawert nicht als integraler Wert darstellen, ist das Ergebnis wieder undefiniert und abhängig davon, was der Compiler damit macht.

```
float fwert = 3.14567;
// Gleitkomma-Integral-Umwandlung -> aus 3.14567 wird 3
int iwert = fwert;
```

Der umgekehrte Fall, wenn ein integraler Typ in einen Gleitkommatyp umgewandelt bzw. diesem zugewiesen wird, ist weniger problematisch. Das Ergebnis entspricht dann einer Gleitkomma-Repräsentation.

Bool-Typumwandlung

Jeder Wert vom integralen Typ, vom Enumerations- oder vom Zeigertyp kann automatisch in einen `bool`-Wert umgewandelt werden. 0 wird somit zum `bool`-Wert `false` und alles andere zum `bool`-Wert `true`.

```
int iwert1 = 100;
int iwert2 = 0;
int* iptr = 0;

// integraler Typ zu bool aus 100 wird true (1)
bool bwert = iwert1;
if( bwert == true ) {
   cout << "iwert1 ist nicht 0!\n";
}

// integraler Typ zu bool aus 0 wird false (0)
bwert = iwert2;
if( bwert == false ) {
   cout << "iwert2 ist 0!\n";
}

// Zeigertyp zu bool aus Nullzeiger wird false (0)
bwert = iptr;
if( bwert == false ) {
   cout << "iptr ist ungültig\n";
}

iptr = &iwert1;
// Zeigertyp zu bool aus gültiger Adresse wird true (1)
bwert = iptr;
if( bwert == true ) {
   cout << "iptr ist gültig\n";
}
```

Zeiger- und Basisklassen-Typumwandlung

Es gibt auch zwei Fälle der automatischen Typumwandlung von Zeigern, und zwar kann jeder Zeigerwert, der den Wert `0` enthält, auch in einen Nullzeiger (`NULL`) umgewandelt werden. Außerdem kann jeder Zeiger eines bestimmten Typs in einen Zeiger vom Typ `void` umgewandelt werden. Hierbei erhält man ebenfalls die Anfangsadresse des Speicherobjekts.

Auch bei den Klassen gibt es die Möglichkeit von Typumwandlungen im Zusammenhang mit der Vererbung. Aber hierauf wird erst in Abschnitt 4.7.7, »Typumwandlung abgeleiteter Klassen«, eingegangen.

3.7.2 Explizite Typumwandlung

Nachdem Sie erfahren haben, wie der Compiler automatisch implizite Umwandlungen vornimmt, sehen Sie jetzt, wie Sie als Programmierer selbst solche Typumwandlungen realisieren können. Sie sollten sich allerdings hierbei immer vor Augen halten, dass Sie den Compiler damit »zwingen«, etwas zu tun, was dieser eigentlich nicht so machen würde. Somit sollte auch klar sein, dass eine gedankenlose Typumwandlung das eine oder andere Problem mit sich bringt. Dies beginnt bei falschen Ergebnissen und kann bis zum Absturz des Programms führen.

[»] **Hinweis**

Bevor Sie sich auf das Abenteuer der expliziten Typumwandlung einlassen, sollten Sie sich überlegen, ob dies nicht zunächst mit der impliziten Umwandlung möglich ist. Das heißt, Sie sollten nur eine explizite Typumwandlung vornehmen, wenn dies nicht mehr anders möglich ist.

Typumwandlung mit dem C-Cast

Zwar werden Sie in der Praxis kaum noch eine Typumwandlung mit den C-Casts machen, aber da es noch viel C-Code (oder eine Mischung aus C und C++) gibt, sollten Sie auch den C-Cast-Operator und dessen Verwendung kennen. Die Syntax dazu ist recht einfach:

```
(Typ) Ausdruck;
```

Hiermit wandeln Sie das Ergebnis des Ausdrucks in den entsprechenden Typ um:

```
float fwert;
int iwert1 = 100, iwert2 = 33;
// Ergebnis der Division von Ganzzahlen
// umwandeln in Gleitpunktzahlen
fwert = (float) iwert1 / iwert2;
```

Ohne die Typumwandlung mit dem C-Cast `(float)` würden Sie als Ergebnis den Wert `3` zurückbekommen. Die Nachkommastellen würden abgeschnitten werden. Was ist also so »schlimm« am C-Cast-Operator? Verwenden wir hierfür doch mal folgendes Beispiel:

```
float fwert;
int iwert1 = 100, iwert2 = 33;
```

```
// Ergebnis der Division von Ganzzahlen
// umwandeln in Gleitpunktzahlen
fwert = (char) iwert1 / iwert2;
```

Eigentlich handelt es sich um dasselbe Beispiel, nur habe ich hier statt eines Casts nach `float` einen Cast nach `char` gemacht – sinnlos, aber möglich.

Leider ist ein C-Cast auch möglich, wenn das Ergebnis des rechten Ausdrucks größer als die Zielgruppe ist. Wird zum Beispiel versucht, eine `float`-Berechnung in einem `short`-Wert mit Hilfe eines C-Casts zu speichern, so lässt sich dies auch »mit Gewalt« realisieren:

```
short swert;
float fwert1 = 1000000, fwert2 = 33;
swert = (short) fwert1 / fwert2;
```

Der Compiler dürfte (sollte) hier zwar eine Warnung ausgeben, aber leider gibt es immer wieder Programmierer, die das ignorieren.

Also ist das Problem an den C-Casts, dass man fast jeden beliebigen Datentyp ohne Sicherheitsabfrage umwandeln kann.

C++ hat hierzu noch eine alternative Notation des alten C-Casts mit eingeführt:

```
Typ (Ausdruck);
```

Diese Notation stellt eine lesbare Alternative zu den alten C-Casts dar und wird wie ein Funktionsstil ausgeführt und auch so bezeichnet:

```
float fwert;
int iwert1 = 100, iwert2 = 33;
// Ergebnis der Division von Ganzzahlen
// umwandeln in Gleitpunktzahlen
fwert = float(iwert1) / float(iwert2);
```

Hinweis [«]

Man kann es nicht oft genug erwähnen: Wenn Sie ein C++-Projekt starten, sollten Sie die neuen Operatoren zur Typumwandlung verwenden.

Neue C++-Typumwandlungs-Operatoren

In C++ wurden vier neue Operatoren zur Typumwandlung eingeführt, deren Verwendung erheblich sicherer ist als die von C-Casts (aber zunächst auch komplizierter). Hier die vier neuen Cast-Operatoren:

```
const_cast<TYP>(ausdruck)
static_cast<TYP>(ausdruck)
```

```
dynamic_cast<TYP>(ausdruck)
reinterpret_cast<TYP>(ausdruck)
```

`TYP` ist hier der neue Typ, in den Sie den Ausdruck konvertieren wollen.

const_cast<TYP>-Typumwandlung

Mit der Typumwandlung `const_cast<TYP>` wandeln Sie einen Typ mit den Qualifikatoren `const` oder `volatile` in einen Ausdruck desselben Typs um, nur ohne den Qualifizierer – der Qualifizierer (`const` oder `volatile`) wird also vorübergehend entfernt.

Ein einfaches Beispiel: Eine Funktion gibt einen `const char*`-Wert zurück, Sie wollen (oder können) den Rückgabewert aber nicht in einem `const char*`-Zeiger speichern, sondern in einem einfachen `char`-Zeiger. Hierzu müssen Sie mit einem `const_cast` vorübergehend den Qualifizierer außer Kraft setzen. Das folgende Listing demonstriert ein solches Beispiel. Die Funktion `substring` sucht im String `s1` nach einem Unterstring `s2` und gibt die Anfangsadresse auf diesen Unterstring zurück (falls vorhanden). Ohne den `const_cast` würde sich das Programm nicht übersetzen lassen.

```
// const_cast.cpp
#include <iostream>
#include <cstring>
using namespace std;

const char* substr( const char* s1, const char* s2 );

int main(void) {
   const char* cptr1 = "Hallo Welt!";
   const char* cptr2 = "Welt";
   // const char* nach char* konvertieren
   char *retptr = const_cast<char*>( substr(cptr1, cptr2) );
   if ( retptr != 0 )  {
      cout << retptr << '\n';    // Welt!
   }
   return 0;
}

const char* substr( const char* s1, const char* s2 ) {
   return strstr( s1, s2 );
}
```

Natürlich bedeutet das jetzt nicht, dass Sie `const_cast` auch auf echte Objekte anwenden können, die bei ihrer Deklaration als `const` vereinbart wurden. Der

Grund ist ganz einfach, da manche Compiler bei einem Typ, den Sie mit `const` qualifizieren, bestimmte Optimierungen vornehmen. Beispielsweise könnte der Compiler diese Konstante in einem Read-only-Speicher wie Flash oder EEPROM ablegen. Ein schreibender Zugriff darauf hat wieder undefinierte Folgen. Folgendes ist also recht kritisch zu betrachten:

```
const int ciwert = 100;
// Nicht erlaubt !!!
int iwert = const_cast<int>(ciwert);
```

Die meisten Compiler sollten hierbei sowieso eine Fehlermeldung ausgeben.

static_cast<TYP>-Typumwandlung

Ein `static_cast<TYP>` können Sie überall einsetzen, um zum Beispiel Ganzzahltypen und Gleitkommatypen hin- und herzukonvertieren. Natürlich bedeutet dies auch wieder, dass zum Beispiel bei einer Konvertierung von `int` nach `char` nur die Bits mit niedrigen Werten übernommen werden. Auch hier wird der Rest wieder verworfen.

```
double dwert = 9.1;
// explizite Typumwandlung von double nach int
int iwert = static_cast<int>(dwert);

// explizite Typumwandlung von int nach float
float fwert = static_cast<float>(iwert);

// Konvertiert das Ergebnis in einen int-Wert
int idoppelt = static_cast<int>( dwert * dwert );
```

Der `static_cast`-Operator wird auch dazu verwendet, um typenlose Zeiger (`void*`) in einen beliebigen anderen Zeiger zu konvertieren, beispielsweise so:

```
void *vptr = ....;
// Zeiger-Umwandlung von void* nach char*
char* cptr = static_cast<char*>(vptr);
```

Natürlich kann der `static_cast`-Operator auch zur definierten Umwandlung von Objekten (genauer Objektzeigern) einer Klasse auf Objekte einer Basisklasse (die in einer Beziehung zueinander stehen) verwendet werden.

reinterpret_cast<TYP>-Typumwandlung

Die Typumwandlungen, die Sie nicht mehr mit `const_cast` oder `static_cast` realisieren können, müssen Sie mit `reinterpret_cast` durchführen. Der `reinterpret_cast` ist also für alle anderen Fälle gedacht. Wenn Sie sich die

Fälle ansehen, die `const_cast` und `static_cast` bereits abdecken, so scheint `reinterpret_cast` für die »exotischeren« Umwandlungen zuständig zu sein. Vorwiegend wird dieser Operator dazu verwendet, verschiedene Zeigertypen und Ganzzahlen in Zeiger bzw. umgekehrt zu konvertieren. Beachten Sie, dass bei einer solchen Umwandlung auch die Bit-Kette anders interpretiert wird, was zu einem komplett neuen Resultat führen kann.

Dieser Operator gibt eine alte Sichtweise aus der C-Programmiersprache wieder, die eine Variable nur noch als Ansammlung von Bytes betrachtet, die man beliebig interpretieren kann. Wenn Sie diesen Operator verwenden, machen Sie zumindest deutlich, dass Sie sich der Risiken bewusst sind. Ein solches Beispiel ist im Umgang mit Dateien gegeben, die im Binärformat gespeichert sind. Wenn Sie den Inhalt einer solchen Datei einlesen, verwenden Sie gewöhnlich die Methode `istream::read()` aus der Standardbibliothek:

```
istream::read(char *ptr, streamsize n);
```

Als erstes Argument müssen Sie einen `char`-Zeiger auf den Speicher übergeben, in dem der Dateiinhalt geschrieben werden soll. Wenn Sie allerdings eine Reihe von `int`-Werten binär einlesen wollen, benötigen Sie einen `reinterpret_cast`, um den Zeiger »umzuinterpretieren«. Hier ein solcher Codeausschnitt:

```
int* buffer;
...
// Puffer in der Größe reservieren
buffer = new int[size];
...
// Datei zuvor noch öffnen
ifstream in(dateiname_zum_lesen, ios::in | ios::binary);
...
// Daten lesen
in.read(reinterpret_cast<char*>(buffer), size*sizeof(int));
```

Beachten Sie bitte außerdem, dass Sie eine teilweise plattformabhängige Typumwandlung vornehmen.

dynamic_cast<TYP>-Typumwandlung

Die letzte Art der Typumwandlung, der `dynamic_cast`-Operator, steht in einem sehr engen Zusammenhang mit dem Typsystem von C++ (RTTI = *Runtime Type Information System*) und wird auch erst in Kapitel 4, »Objektorientierte Programmierung«, wo die Klassen und Vererbungen näher behandelt werden, erläutert, da dieser Operator auch nur in diesem Zusammenhang von Bedeutung ist.

Häufig wurden Sie bereits auf dieses Kapitel verwiesen. Hier wird jetzt auf moderne objektorientierte Programmierung eingegangen, die sich hervorragend dazu eignet, auf hohem Abstraktionsniveau die Komplexität umfangreicher Projekte zu realisieren.

4 Objektorientierte Programmierung

4.1 OOP-Konzept versus prozedurales Konzept

Bevor wir uns mit der objektorientierten Programmierung auseinandersetzen, soll nochmals das »neue« OOP-Konzept mit dem »alten« prozeduralen Konzept, wie es zum Beispiel in der Programmiersprache C verwendet wird, verglichen werden.

Bei der klassischen prozeduralen Programmierung war es immer so, dass Daten und Funktionen keine Einheit bildeten (siehe Abbildung 4.1). Probleme traten hierbei häufiger bei falschem Zugriff auf (z. B. nicht initialisierte) Daten auf. Musste dann noch ein Programm umgeschrieben bzw. erweitert werden, war der (Zeit-)Aufwand häufig enorm, und weitere Fehler waren so gut wie sicher, wie es die Praxis in der Vergangenheit immer wieder gezeigt hat.

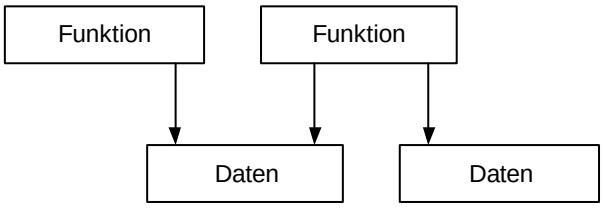

Abbildung 4.1 Klassisches prozedurales Konzept

Bei der objektorientierten Programmierung hingegen bilden die Objekte eine echte Einheit aus Daten und Funktionen – wobei in der OOP die Daten als *Eigenschaften* und die Funktionen als *Methode* bezeichnet werden (siehe Abbildung 4.2).

Der Vorteil des OOP-Konzepts ist für einen Anfänger bzw. Umsteiger zunächst noch nicht nachvollziehbar, dies wird sich aber auf den nächsten Seiten noch ändern. Es lässt sich allerdings nicht vermeiden, dass die Komplexität im Verlauf des Kapitels

immer mehr zunimmt. Man sollte sich also Zeit nehmen, die OOP-Paradigmen zu verstehen – es lohnt sich auf jeden Fall. Anschließend wird Ihnen auch das Erlernen von Java oder C# bzw. jeder anderen OOP-Sprache leichter fallen.

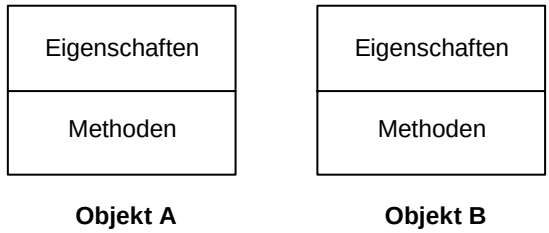

Abbildung 4.2 Objektorientiertes Konzept

4.1.1 OOP-Paradigmen

Unabhängig von der Programmiersprache bieten alle OOP-Programmiersprachen zur vollen Unterstützung folgende OOP-Paradigmen an:

- Datenabstraktion – es werden fortgeschrittene Datentypen (in diesem Fall *Klassen*) definiert, die die Eigenschaften (Daten) und Methoden (Funktionen) von Objekten beschreiben – oder einfach ausgedrückt: Wenn man einen Typ `int` für Ganzzahlen oder ein `double` für Gleitpunktzahlen verwenden kann, warum sollte man dann nicht einen Typ `Festplatte` mit seinen Fähigkeiten definieren und verwenden.
- Datenkapselung – einzelne Elemente eines Objekts einer Klasse können vor falschem Zugriff von außen geschützt werden. Zum Datenaustausch mit anderen Objekten besitzt jedes Objekt eine »offene« Schnittstelle – oder einfach ausgedrückt: Wenn Sie einen Typ `Festplatte` haben und verwenden wollen, brauchen Sie nicht zu wissen, wie diese funktioniert.
- Vererbung – neue Objekte können aus bereits vorhandenen Objekten abgeleitet werden. Damit »erbt« das neue Objekt Eigenschaften (Daten) und Methoden (Funktionen) des vorhandenen Objekts – oder einfach ausgedrückt: Wenn Sie schon ein Objekt `Festplatte` haben, können Sie auch ein Objekt `DVD-Brenner`, `CD-Brenner` usw. mit den Methoden (Funktionen) `Daten lesen`, `Daten suchen`, `Daten schreiben` usw. sowie den Eigenschaften (Daten) `Schnittstelle` (z. B. S-ATA oder SCSI), `Datendurchsatz`, `Farbe` usw. beerben und müssen diese Angaben nicht immer wieder neu schreiben.
- Polymorphie – die Vererbung ist auch auf ganze Familien gleicher Objekte anwendbar. Je nach Typ des Objekts kann der Aufruf zur Laufzeit dann eine andere Eigenschaft (Daten) verwenden oder eine andere Methode aufrufen – oder einfach ausgedrückt: Wenn Sie ein Objekt `Tier` definiert haben, dann

haben viele Tiere eine ähnliche Eigenschaft (Daten), aber die Methoden (Funktionen oder hier besser: Fähigkeiten) dieser Tiere sind oft andere – die Kuh zum Beispiel macht »muh« und nicht »wau« (es sei denn, die Gentechniker arbeiten dran).

Auch wenn sich die objektorientierte Programmierung in der Theorie erschreckend kompliziert anhört – in der Praxis erweist sie sich als sehr einleuchtend.

Hinweis **[«]**

Vielleicht sind Sie manches Mal etwas genervt, wenn die Autoren der Bücher immer als Objekte einfache Dinge aus dem Leben verwenden (wie Autos, Tiere, Lebewesen etc.). Dies ist häufig der einfachste Weg, die OOP dem Leser näherzubringen. Natürlich könnte man als Objekt auch einen Typ `Otto-Motor` oder `Kernreaktor` definieren und verwenden, aber dies setzt dann noch weitere (Fach-)Kenntnisse auf diesen Gebieten voraus.

4.2 Klassen (fortgeschrittene Typen)

Dieser Abschnitt schließt an Abschnitt 2.8, »Fortgeschrittene Typen«, an. Zunächst werden Sie überrascht und vielleicht auch enttäuscht sein, dass Klassen nichts anderes als einfache Strukturen (`struct`) sind, die abgesehen davon auch Funktionen (Methoden) enthalten dürfen und üblicherweise das Schlüsselwort `class` statt `struct` verwenden. Wenn Sie den Abschnitt zu den Strukturen komplett gelesen haben, wird Ihnen der Einstieg in die Welt der Klassen leichter fallen.

Hinweis **[«]**

Zwar wird gewöhnlich das Schlüsselwort `class` für die Klassen verwendet, aber Sie können ohne weiteres auch das Schlüsselwort `struct` benutzen. Wenn Sie in Ihrer Klasse nicht das Schlüsselwort `public` oder `private` verwenden, besteht der Unterschied zwischen `struct` und `class` darin, dass bei einer mit `struct` definierten Klasse alle Daten `public` (also öffentlich zugänglich) sind und beim Schlüsselwort `class` alle Daten `private` (nur zur Klasse gehörende Daten). Auf `public` und `private` wird noch eingegangen.

Der Begriff *Klasse* ist zunächst ein wenig unklar. Deshalb stellen Sie sich einmal vor, wie Sie spazieren gehen und sich dabei umsehen. Sie nehmen Bäume, Häuser, Menschen, Hunde und Autos wahr. Ihr erster Blick fällt vielleicht auf einen Baum. Wenn Sie diesen Baum aber etwas länger betrachten, werden Sie beginnen, ihn zu klassifizieren. Jeder kann noch zwischen einem Nadelbaum und einem Laubbaum unterscheiden (zwei Klassen der Oberklasse *Baum*). Wer jetzt auch noch einige botanische Kenntnisse hat, kann die einzelnen Bäume noch

genauer klassifizieren (z. B. Birke, weiße Baumrinde, grüne Blätter, ca. 20 Meter hoch usw.). Ebenso verläuft dies bei einem Menschen, den Sie sehen – ein Lebewesen der Gattung Homo sapiens. Menschen unterscheiden sich durch die Haarfarbe, die Augenfarbe, die Hautfarbe, die Größe usw. – so oder so ähnlich geht das menschliche Gehirn vor. Was zunächst unbedeutend erscheint, ergibt bei näherem Betrachten jedoch mehr Sinn.

4.2.1 Klassen deklarieren

Wir bleiben bei unserer Spezies, dem Homo sapiens, und spielen ein wenig Gott – oder denken Sie, Gott verwendet noch eine prozedurale Sprache? Zunächst benötigen Sie einen Rahmen für eine neue Klasse. Eine leere Klasse `Mensch` sieht demnach wie folgt aus:

```
class Mensch {
   // Hier kommen die Eigenschaften und Fähigkeiten hin
};
```

Vergessen Sie das Semikolon am Ende dieser Klassendeklaration, kann dies zu seltsamen Fehlermeldungen führen.

Für den Klassennamen gelten dieselben Regeln wie bei den Bezeichnern (siehe Abschnitt 1.3.1, »Bezeichner«). Allerdings sollte ein Klassenname immer eindeutig sein. Gewöhnlich verwendet man einen großen Anfangsbuchstaben, was zwar kein Standard ist, aber gängige Praxis.

Als Nächstes werden die Elemente (auch *Member* genannt) zur Klasse hinzugefügt. Zu den Elementen (Member) einer Klasse gehören die Eigenschaften (Daten) und die Methoden bzw. Fähigkeiten (Funktionen), die zwischen den Anweisungsblock der geschweiften Klammern einer Klasse geschrieben werden.

```
class Mensch {
   char name[30];
   unsigned int alter;
   bool geschlecht; //0 = männlich; 1 = weiblich
};
```

Hier haben Sie die Klasse `Mensch` mit den Eigenschaften Name, Alter und Geschlecht versehen. In der Realität hat ein Mensch natürlich mehr Eigenschaften, aber hier sollte das zunächst genügen. Die Eigenschaften der Klasse werden auch als *Klassendaten* bezeichnet.

Die Eigenschaften allein machen noch keine Klasse aus, und das Beschriebene entspricht immer noch dem Niveau einer einfachen Struktur. Die Klasse (hier

Mensch) besitzt im Grunde immer noch einige Fähigkeiten (Funktionen). Auch hierbei habe ich mich auf einige grundlegende Fähigkeiten beschränkt:

```
class Mensch {
   // Eigenschaften der Klasse Mensch
   char name[30];
   unsigned int alter;
   //0 = männlich; 1 = weiblich
   bool geschlecht;

   // Fähigkeiten (Methoden) der Klasse Mensch
   void sehen( const char* objekt );
   void hoeren( const char* geraeusch );
   void riechen( const char* geruch );
   void schmecken( const char* geschmack );
   void tasten( const char* objekt );
   // Einen Menschen mit allen Daten erzeugen
   void erzeuge( const char* n, unsigned int a, bool g );
   void print( void );
   void test_geschlecht( void );
};
```

Hier haben wir uns bei der Deklaration der Klasse Mensch auf die Fähigkeiten der fünf Sinne beschränkt. Natürlich haben Sie mit dieser Klasse Mensch noch lange keinen Speicher reserviert – es handelt sich lediglich um eine Anweisung für den Rechner, was die Klasse Mensch alles darstellt. Der Rechner weiß hierbei, wie viel Speicherplatz er für ein Objekt Mensch reservieren muss. Allerdings wird nur wieder Speicher für die Eigenschaften (Daten) reserviert. Wie Sie die Elementfunktionen (Methoden, Fähigkeiten) definieren können, erfahren Sie weiter unten.

4.2.2 Elementfunktion (Klassenmethode) definieren

Die Methoden (Funktionen), die Sie in der Klasse deklariert haben, müssen Sie nun auch definieren, um eine Klassendefinition vollständig zu machen. Theoretisch könnten Sie die Definition einer Elementfunktion auch gleich in der Klasse selbst vornehmen, aber in der Praxis ist dies doch eher unüblich:

```
class Mensch {
   // Eigenschaften der Klasse Mensch
   char name[30];
   unsigned int alter;
   //0 = männlich; 1 = weiblich
   bool geschlecht;
```

```
   // Fähigkeiten (Methoden) der Klasse Mensch
   // Definition der Elementfunktion sehen in der Klasse
   void sehen( const char* objekt ) {
      // Anweisungen für die Funktion
   }
};
```

In der Praxis wird wegen der Übersichtlichkeit die Definition einer Elementfunktion im Allgemeinen außerhalb der Klasse vorgenommen. Allerdings genügt es dann nicht mehr, bei der Definition nur den Funktionsnamen anzugeben. Hierbei müssen Sie also zunächst den Klassennamen, gefolgt vom Scope-Operator (Bereichsoperator) `::`, und dann den eigentlichen Funktionsnamen verwenden. Die Syntax hierzu lautet:

```
Typ Klassenname::funktionsname( parameter ) {
   // Anweisungen der Funktion
}
```

So teilen Sie dem Compiler mit, dass die Funktion `funktionsname` eine Methode (Fähigkeit) der Klasse `Klassenname` ist. Würden Sie den Klassennamen nicht verwenden, so hätten Sie nur eine einfache globale Funktion definiert. Bezogen auf die Klasse `Mensch` und die Elementfunktion `sehen`, sieht eine Definition außerhalb der Klasse wie folgt aus:

```
void Mensch::sehen( const char* objekt ) {
     // Anweisungen
}
```

Die Definition der Elementfunktion `sehen()` sieht bei der Klasse `Mensch` wie folgt aus:

```
void Mensch::sehen( const char* objekt ) {
   cout << name << " sieht " << objekt << "\n";
}
```

Hier können Sie auch gleich erkennen, dass Sie auf die Eigenschaften (Daten) einer Klasse innerhalb der Methode direkt zugreifen können – also ohne irgendwelche Scope-Operatoren. Dies funktioniert natürlich auch, wenn Methoden einer Klasse andere Methoden derselben Klasse aufrufen. Auch hierbei kann ohne weiteren Bezug auf diese Klasse zugegriffen werden. Somit kann man sagen, dass alle Eigenschaften und Methoden innerhalb einer Klasse ohne besonderen Bezug zueinander harmonieren, so dass Elementfunktionen (Methoden) ohne Umweg auf die Eigenschaften (Daten) und andere Elementfunktionen zugreifen können.

4.2.3 Objekte deklarieren

Jetzt haben Sie zwar eine Klasse `Mensch` geschaffen, aber eine solche Klasse entspricht eher der Vorstellung eines Menschen. Und Gedanken an einen Menschen machen diesen noch nicht real.

Sie müssen also ein Objekt (auch als *Instanz* einer Klasse bezeichnet) deklarieren. Klassen und Objekte? Bevor Sie etwas durcheinanderbringen: Eine Klasse ist zunächst der bloße Gedanke an einen Gegenstand. Wenn wir von einem Objekt selbst reden, dann handelt es sich um einen realen Gegenstand, der sich vor Ihren Augen befindet (natürlich nicht wirklich, aber von der IT-Welt auf unser Gehirn projiziert). Die Deklaration eines Objekts verläuft wie eine beliebige andere Typdeklaration:

```
Klassenname Objekt;
```

Auf unser Beispiel »Mensch« bezogen, sieht dies wie folgt aus:

```
Mensch frau;
```

Natürlich können Sie auch mehrere Objekte deklarieren:

```
Mensch frau, mann;
```

Mit einer solchen Deklaration wird jetzt Speicher für die Eigenschaften (Daten) des Objekts `frau` oder `mann` reserviert. Im Beispiel sind dies die Datenelemente `name`, `alter` und `geschlecht`. Natürlich gilt dies für jedes Objekt.

Allerdings arbeiten beide Objekte mit denselben Methoden, mit denen der Code dieser Klassenmethoden nur einmal im Speicher abgelegt wird – und das ist wiederum unabhängig davon, ob und wie oft ein Objekt dieser Klasse existiert.

Wie auch bei den anderen Typen, die Sie bisher kennen, ist der Inhalt der Daten zunächst undefiniert. Wenn Sie das Objekt als `static` oder gar `global` (nicht ratsam) definieren, so wird der Inhalt auch hier standardmäßig mit `0` belegt.

4.2.4 Kurze Zusammenfassung

Eine Klasse besitzt mehrere Elemente (auch als Member bezeichnet). Diese Elemente (Member) einer Klasse werden aufgeteilt in deren Eigenschaften (die Daten) und die Methoden bzw. Fähigkeiten (Member-Funktionen oder auch Elementfunktionen). Diese Elementfunktionen sind in der Regel die offenen Schnittstellen (Interfaces) einer Klasse nach außen. Über diese Funktionen werden die Eigenschaften (Daten) einer Klasse angesprochen. Wenn eine Klasse erzeugt wurde, kann ein Objekt erzeugt werden. Ein Objekt wird auch als Instanz einer Klasse bezeichnet. Hier nochmals die allgemeine Syntax dazu:

```
class Klassenname {
   // Zusammenfassung der Klassenelemente - Anfang

   // Eigenschaften einer Klasse
   typ daten1;
   typ daten2;
   ...
   // Methoden bzw. Fähigkeiten einer Klasse
   typ funktionsname1( parameter );
   typ funktionsname2( parameter );
   ...

   // Zusammenfassung der Klassenelemente - Ende
};

// Definitionen der Methoden
Typ Klassenname::funktionsname( parameter ) {
   // Anweisungen der Funktion
}

...
// Objekt deklarieren - Instanz einer Klasse
Klassenname objektname;
```

4.2.5 »private« und »public« – Zugriffsrechte in der Klasse

Mit dem bisherigen Wissen über Klassen kann noch kein funktionierender Quellcode erstellt werden, denn bei der Deklaration von Klassen müssen auch die Zugriffsrechte auf deren Elemente mit den Schlüsselwörtern `public` oder `private` erteilt werden. Bisher sieht unsere Klasse `Mensch` wie folgt aus:

```
class Mensch {
   // Eigenschaften der Klasse Mensch
   char name[30];
   unsigned int alter;
   //0 = männlich; 1 = weiblich
   bool geschlecht;

   // Fähigkeiten (Methoden) der Klasse Mensch
   void sehen( const char* objekt );
   void hoeren( const char* geraeusch );
   void riechen( const char* geruch );
   void schmecken( const char* geschmack );
   void tasten( const char* objekt );
   // Einen Menschen mit allen Daten erzeugen
```

```
   void erzeuge( const char* n, unsigned int a, bool g );
   void print( void );
};
```

Würden Sie jetzt ein Objekt deklarieren und versuchen, auf die Klassenelemente zuzugreifen, würde sich das Programm nicht übersetzen lassen. Der Grund sind die Standardzugriffsrechte, die jede Klasse ohne explizite Angaben besitzt. Standardmäßig ist der Zugriff bei Klassen – im Gegensatz zu Strukturen (`struct`) – von außen verboten. Der Grund solcher Rechte ist, dass der Zugriff auf die Elemente einer Klasse nur noch kontrolliert erfolgen soll. So werden Probleme wie die falsche Übergabe von Argumenten oder nicht initialisierte Variablen bei richtiger Anwendung ausgeschlossen.

Der Standardzugriff aller Elemente einer Klasse ist `private`. Dies bedeutet, die Eigenschaften und Methoden einer Klasse können nur innerhalb der Klasse angesprochen werden. Die »Sperrung« gilt außerhalb der Klasse. Methoden einer Klasse können immer auf die Eigenschaften der eigenen Klasse zurückgreifen. Es leuchtet aber ein, dass die Standardeinstellung `private` allein für die Verwendung von Klassen sinnlos sind, da überhaupt nicht von außen auf die Elemente einer Klasse zugegriffen werden kann. Aus diesem Grunde muss mindestens für ein Element die Voreinstellung aufgehoben werden, um die Klasse von außen ansprechen zu können.

Aufheben kann man eine solche Sperre mit dem Schlüsselwort `public`. Alle Elemente, die das Zugriffsrecht `public` haben, können von außerhalb der Klasse angesprochen werden. Hier die Syntax dazu:

```
class Klassenname {
   // Auf Elemente kann nur innerhalb einer Klasse
   // zugegriffen werden
   private:
   // Eigenschaften einer Klasse
   typ daten1;
   typ daten2;
   ...

   // Zugriff von außen auf die Elemente möglich
   public:
   // Methoden bzw. Fähigkeiten einer Klasse
   typ funktionsname1( parameter );
   typ funktionsname2( parameter );
   ...

};
```

Alle Elemente, die hier hinter `private` (Doppelpunkt muss angegeben werden) stehen, sind nur innerhalb der Klasse »sichtbar«. Von außen kann auf sie nicht zugegriffen werden. Diese Rechteerteilung gilt bis zum Ende der Klasse – oder bis zum Schlüsselwort `public` (auch hier ist der Doppelpunkt dahinter wichtig). Ab dem Schlüsselwort `public` ist alles dahinter Geschriebene »öffentlich« außerhalb der Klasse zugänglich. Es ist im Grunde egal, ob Sie zuerst die Klassenelemente `public` oder `private` oder beide gemischt verwenden. Es ist auch egal, wie oft Sie `public` oder `private` in einer Klasse verwenden und ob Sie dabei Eigenschaften und/oder Methoden öffentlich oder privat zugänglich machen.

In der Praxis werden aber gewöhnlich als Entwurfsmuster die Eigenschaften einer Klasse als `private` vorbehalten und die Methoden (Funktionen) als `public` (öffentlich). Damit lässt sich der Zugriff auf die Daten (Eigenschaften) über die Schnittstellen (Elementfunktionen) kontrollieren. Um auf die Daten einer Klasse zuzugreifen, wird eine sogenannte Zugriffsmethode (Funktion) geschrieben. Hiermit wird praktisch auf die privaten Eigenschaften einer Klasse zugegriffen. In der Klasse `Mensch` ist dies im Beispiel die Zugriffsmethode `erzeuge()`.

Hiermit sieht unsere Klasse `Mensch` bisher wie folgt aus:

```
class Mensch {
private:
   // Eigenschaften der Klasse Mensch
   char name[30];
   unsigned int alter;
   bool geschlecht; //0 = männlich; 1 = weiblich
   // Methode nur innerhalb der Klasse ansprechbar
   void test_geschlecht( void );

public:
   // Fähigkeiten (Methoden) der Klasse Mensch
   void sehen( const char* objekt );
   void hoeren( const char* geraeusch );
   void riechen( const char* geruch );
   void schmecken( const char* geschmack );
   void tasten( const char* objekt );
   // Einen Menschen mit allen Daten erzeugen
   void erzeuge( const char* n, unsigned int a, bool g );
   void print( void );
};
```

4.2.6 Zugriff auf die Elemente (Member) einer Klasse

Wie Sie bereits erfahren haben, können Sie von außerhalb der Klasse nur auf die `public`-Elemente zugreifen. Eine Ausnahme hierbei ist, wie auch bereits

erwähnt, wenn Sie auf die Eigenschaften einer Klasse mit einer Elementfunktion derselben Klasse zugreifen wollen. Dann sind keine Besonderheiten zu beachten. Hier benötigen Sie weder den Scope-Operator noch den Punkt- bzw. Pfeiloperator. Es ist gängige Praxis, die Eigenschaften (Daten) einer Klasse nicht direkt, sondern immer nur über eine Elementfunktion (Zugriffsmethode) anzusprechen. Hierzu wird in der Klasse `Mensch` die Funktion `erzeuge()` als Zugriffsmethode verwendet:

```
void Mensch::erzeuge(const char* n,unsigned int a,bool g) {
   // Hier könnten/sollten die übergebenen Parameter
   // überprüft werden, bevor diese verwendet werden
   strncpy( name, n, sizeof(name)-1 );
   name[sizeof(name)] = '\0';
   alter = a;
   geschlecht = g;
}
```

Ansonsten erfolgt der Zugriff auf die `public`-Elemente eines Objekts von außen wie schon bei den Strukturen mit dem Punkt- und Pfeiloperator.

Direkter Zugriff mit dem Punktoperator

Wenn Sie ein Objekt definiert haben, können Sie den Punktoperator verwenden, um auf die `public`-Elemente direkt zuzugreifen (wie schon bei den gewöhnlichen Strukturen mit `struct`). Im Grunde können Sie auf die Klassenelemente immer nur mit einem Objekt der Klasse direkt zugreifen. Dieser direkte Zugriff auf ein Element erfolgt durch die Bezeichnung des Objekts, gefolgt vom Punktoperator und dem Bezeichner des Klassenelements. Die Syntax lautet:

```
// Objekt deklarieren
Klassenname Objekt;
// Zugriff auf public-Eigenschaften der Klasse
Objekt.Eigenschaft = ..;
// Zugriff auf public-Methode der Klasse
Objekt.Methode( parameter );
```

Im Beispiel der Klasse `Mensch` können Sie einen Menschen folgendermaßen erschaffen:

```
// Objekt mann der Klasse Mensch
Mensch mann;
// Zugriffsmethoden aufrufen
mann.erzeuge( "Adam", 18, 0 );
```

Im Beispiel wurde gleich die Zugriffsmethode zum Erzeugen eines Menschen verwendet:

```
void Mensch::erzeuge(const char* n,unsigned int a,bool g) {
   // Hier könnten/sollten die übergebenen Parameter
   // überprüft werden, bevor diese verwendet werden
   strncpy( name, n, sizeof(name)-1 );
   name[sizeof(name)] = '\0';
   alter = a;
   geschlecht = g;
}
```

Natürlich können Sie theoretisch auch auf die einzelnen Eigenschaften einer Klasse von außen zugreifen. Aber wie bereits erwähnt, sollte man diese Daten immer als `private`-Elemente kennzeichnen, so dass eine Methode dafür verantwortlich ist, dass die richtigen Daten eingegeben werden. Bei der Klasse `Mensch` würde folgender Zugriff außerhalb einer Klassenfunktion auf die Eigenschaften zu einem Fehler führen:

```
// Objekt mann der Klasse Mensch
Mensch mann;
// Fehler !!! name ist private
strncpy( mann.name, "Adam", sizeof(mann.name)-1 );
mann.name[sizeof(name)] = '\0';
// Fehler !!! alter ist private
mann.alter = 18;
// Fehler !!! geschlecht ist private
mann.geschlecht = 0;
```

`private`-Elemente haben ihre Bedeutung aber nicht nur beim Beschreiben einer Eigenschaft, sondern auch beim Lesen. So können Sie auch nicht einzelne `private`-Elemente zum Beispiel mit `cout` auf dem Bildschirm ausgeben:

```
// Fehler !!! name ist private
cout << mann.name;
```

Wollen Sie trotzdem direkt auf die Eigenschaften der Klasse `Mensch` über das Objekt `mann` zugreifen, müssten Sie diese Daten als `public`-Elemente kennzeichnen, was aber aus Designgründen der Sicherheit vermieden werden soll. Dadurch würden die Vorteile der OOP wieder aufgehoben. Mit folgender Klassendeklaration könnten Sie auf alle Elemente (Eigenschaften und Methoden) von außen zugreifen:

```
class Mensch {
// Alle Klassenelemente nach außen sichtbar ->
// -> schlechtes Programmdesign
public:
   // Eigenschaften der Klasse Mensch
   char name[30];
```

```
   unsigned int alter;
   bool geschlecht; //0 = männlich; 1 = weiblich
   void test_geschlecht( void );
   // Fähigkeiten (Methoden) der Klasse Mensch
   void sehen( const char* objekt );
   void hoeren( const char* geraeusch );
   void riechen( const char* geruch );
   void schmecken( const char* geschmack );
   void tasten( const char* objekt );
   // Einen Menschen mit allen Daten erzeugen
   void erzeuge( const char* n, unsigned int a, bool g );
   void print( void );
};
```

Ein zu Anfang häufig gemachter Fehler bei der Verwendung von Objekten auf Klassen ist der, dass vergessen wird, das Objekt anzugeben, über das eine Klassenmethode aufgerufen werden soll:

```
// Fehler !!! Sucht nach einer globalen Funktion erzeuge()
erzeuge( "Adam", 18, 0 );
```

Wenn es hierbei keine globale Funktion `erzeuge()` gibt, führt dieser Funktionsaufruf zu einem Fehler beim Übersetzen des Quellcodes.

Indirekter Zugriff mit dem Pfeiloperator

Wie schon bei den Zeigern auf Strukturen können Sie auch mit Zeigern auf Objekte von Klassen arbeiten – was bei dynamischen Datenstrukturen durchaus gängig ist. Der indirekte Zugriff wird mit dem Objektzeiger, gefolgt vom Pfeiloperator und dem Bezeichner eines Elements (Eigenschaft oder Methode), realisiert. Natürlich muss man auch hier vor dem Einsatz des Objektzeigers zunächst auf eine gültige Adresse verweisen, bevor man diesen verwendet. Der Vorteil von Objektzeigern ist auch, dass man diesen jederzeit wieder eine andere Adresse zuweisen kann. Die Syntax hierzu lautet:

```
// Objekt deklarieren
Klassenname Objekt;
// Objektzeiger deklarieren
Klassenname* ObjektPtr;

// Gültige Adresse an ObjektPtr zuweisen
ObjektPtr = &Objekt;

// Indirekter Zugriff auf public-Eigenschaften der Klasse ->
ObjektPtr->Eigenschaft = ..;
```

```
// Indirekter Zugriff auf public-Methode der Klasse
ObjektPtr->Methode( parameter );
```

Bezogen auf unser Objekt `mann` der Klasse `Mensch` sieht der indirekte Zugriff wie folgt aus:

```
// Objekt mann der Klasse Mensch
Mensch mann;
// Objektzeiger
Mensch* MenschPtr;

// Gültige Adresse an MenschPtr zuweisen
MenschPtr = &mann;
// Zugriffsmethoden aufrufen
MenschPtr->erzeuge( "Adam", 18, 0 );
```

Bevor hierzu ein Listing folgt, das das bisher Beschriebene in der Anwendung demonstriert, soll noch erläutert werden, wie man auf `enum`-Eigenschaften zugreifen kann. Im Beispiel fällt auf, dass hier für die Erzeugung eines Menschen `0` für männlich und `1` für weiblich verwendet wird. Hierzu könnte man einen besser lesbaren `enum`-Typ wie folgt einführen:

```
// 0 ist männlich, 1 ist weiblich
enum Geschlecht { MANN, FRAU };
```

Jetzt kann man natürlich diesen `enum`-Datentyp `global` deklarieren. Somit würde beim Erzeugen von »Adam« Folgendes genügen:

```
// Erzeugt einen Mann
mann.erzeuge( "Adam", 18, MANN );
```

Aber wenn Sie richtig objektorientiert programmieren wollen, dann sollten Sie diesen `enum`-Datentyp auch in der Klasse deklarieren, in die er gehört. Sie sollten ihn als `public` deklarieren, wenn Sie außerhalb der Klassenmethoden darauf zugreifen wollen. Wollen Sie die `enum`-Konstanten außerhalb der Klassenmethoden verwenden, müssen Sie allerdings den Klassennamen und den Scope-Operator mit angeben.

[»] Hinweis

Natürlich sollten Sie hier auch nur den `enum`-Datentyp als `public` deklarieren. `enum`-Eigenschaften sollten möglichst wie alle Klasseneigenschaften als `private` deklariert werden.

Somit sieht der Zugriff des (`public`-)`enum`-Datentyps in der Klasse `Mensch` außerhalb der Klassenmethoden folgendermaßen aus:

```
mann.erzeuge( "Adam", 18, Mensch::MANN );
```

Hierzu nun das komplette Listing, das alle bisher besprochenen Aspekte in der Praxis demonstriert:

```
// class1.cpp
#include <iostream>
#include <cstring>
using namespace std;

class Mensch {
public:
   enum Geschlecht { MANN, FRAU };
private:
   // Eigenschaften der Klasse Mensch
   char name[30];
   unsigned int alter;
   bool geschlecht;       // 0 = männlich; 1 = weiblich
   void test_geschlecht( void );

public:
   // Fähigkeiten (Methoden) der Klasse Mensch
   void sehen( const char* objekt );
   void hoeren( const char* geraeusch );
   void riechen( const char* geruch );
   void schmecken( const char* geschmack );
   void tasten( const char* objekt );
   // Einen Menschen mit allen Daten erzeugen ggf. überladen
   void erzeuge( const char* n = "Unbekannt",
                 unsigned int a = 0,
                 bool g = FRAU );
   void print( void );
};

int main(void) {
   // Mehrere Objekte deklarieren
   Mensch mann, frau, person, dummy;
   // Ein Objektzeiger
   Mensch* Menschptr;

   // Zugriffsmethoden aufrufen
   mann.erzeuge( "Adam", 18, Mensch::MANN );
   frau.erzeuge( "Eva" , 18, Mensch::FRAU );

   // einige Aktionen (Klassenmethoden) aufrufen
   mann.sehen( "Eva" );
   frau.sehen( "Apfel" );
```

```
   frau.tasten( "Apfel" );
   mann.hoeren( "Warnung von Gott" );
   frau.hoeren( "nichts" );
   frau.riechen( "Apfel" );
   frau.schmecken( "Apfel" );
   mann.hoeren( "Schlange" );
   // So gehts auch - dank Standardparametern
   person.erzeuge( );

   // Oder indirekt mit einem Zeiger über den Pfeiloperator
   Menschptr = &dummy;
   Menschptr->erzeuge( "Jürgen Wolf", 30, Mensch::MANN );

   // Ausgabe aller erzeugten Menschen
   cout << "\nMeine erzeugten Menschen bisher : \n";
   mann.print();
   frau.print();
   person.print();
   Menschptr->print();
   return 0;
}

// Hier beginnen die Definitionen der Klassenmethoden

void Mensch::erzeuge(const char* n,unsigned int a,bool g) {
   // Hier könnten/sollten die übergebenen Parameter
   // überprüft werden, bevor diese verwendet werden
   strncpy( name, n, sizeof(name)-1 );
   name[sizeof(name)] = '\0';
   alter = a;
   geschlecht = g;
}

void Mensch::print ( void ) {
   cout << name << " " << alter << " Jahre (";
   test_geschlecht();
   cout << ")\n";
}

void Mensch::test_geschlecht( void ) {
   if( geschlecht == FRAU ) {
      cout << "weiblich";
   }
   else {
      cout << "männlich";
```

```
   }
}

void Mensch::sehen( const char* objekt ) {
   cout << name << " sieht " << objekt << '\n';
}
void Mensch::hoeren( const char* geraeusch ) {
   cout << name << " hört " << geraeusch << '\n';
}

void Mensch::riechen( const char* geruch ) {
   cout << name << " riecht " << geruch << '\n';
}

void Mensch::schmecken( const char* geschmack ) {
   cout << name << " schmeckt " << geschmack << '\n';
}

void Mensch::tasten( const char* objekt ) {
   cout << name << " nimmt " << objekt << '\n';
}
```

Das Programm bei der Ausführung:

```
Adam sieht Eva
Eva sieht Apfel
Eva nimmt Apfel
Adam hört Warnung von Gott
Eva hört nichts
Eva riecht Apfel
Eva schmeckt Apfel
Adam hört Schlange

Meine erzeugten Menschen bisher :
Adam 18 Jahre (männlich)
Eva 18 Jahre (weiblich)
Unbekannt 0 Jahre (weiblich)
Jürgen Wolf 30 Jahre (männlich)
```

4.2.7 Ein Programm organisieren

Bei kleineren Projekten und Programmen, wie sie zum Beispiel in diesem Buch verwendet werden, ist es meistens nicht nötig, das Programm zu organisieren. Aber ein größeres Projekt sollte man immer in mehrere Module aufteilen. Hierbei wird gewöhnlich die Definition in eine separate Header-Datei gepackt. Auch

die Definition der Klassenmethoden sollte man in ein anderes Modul packen – dafür sollte allerdings keine Header-Datei mehr verwendet werden, sondern eine weitere Quelldatei (*.cpp*).

Da Sie bisher nur mit der Klasse `Mensch` gearbeitet haben, soll anhand des folgenden Listings demonstriert werden, wie Sie ein solches Programm sinnvoll organisieren. Zunächst also die Header-Datei *mensch.h*, die zeigt, wie die Klasse definiert wird:

```
// mensch.h
#include <iostream>

#ifndef _MENSCH_H_
#define _MENSCH_H_
using namespace std;

class Mensch {
public:
   enum Geschlecht { MANN, FRAU };
private:
   // Eigenschaften der Klasse Mensch
   char name[30];
   unsigned int alter;
   bool geschlecht;        //0 = männlich; 1 = weiblich
   void test_geschlecht( void );

public:
     // Fähigkeiten (Methoden) der Klasse Mensch
   void sehen( const char* objekt );
   void hoeren( const char* geraeusch );
   void riechen( const char* geruch );
   void schmecken( const char* geschmack );
   void tasten( const char* objekt );
   // Einen Menschen mit allen Daten erzeugen ggf. Überladen
   void erzeuge( const char* n = "Unbekannt",
                 unsigned int a = 0,
                 bool g = FRAU );
   void print( void );
};
#endif
```

Zwar können Sie die Quelldateien und die Header-Dateien benennen, wie es Ihnen gefällt, aber aus ersichtlichen Gründen wird die Header-Datei einer Klasse auch ähnlich wie die Klasse bezeichnet.

Das sollte auch bei der Bezeichnung der Quelldatei mit den Klassenmethoden der Fall sein. Im Beispiel habe ich die Methoden der Klasse in eine Quelldatei namens *mensch.cpp* gepackt und definiert:

```
// mensch.cpp
#include <iostream>
#include <cstring>
#include "mensch.h"
using namespace std;
// Hier beginnen die Definitionen der Klassenmethoden

void Mensch::erzeuge(const char* n, unsigned int a,bool g) {
   // Hier könnten/sollten die übergebenen Parameter
   // überprüft werden, bevor diese verwendet werden
   strncpy( name, n, sizeof(name)-1 );
   name[sizeof(name)] = '\0';
   alter = a;
   geschlecht = g;
}

void Mensch::print ( void ) {
   cout << name << " " << alter << " Jahre (";
   test_geschlecht();
   cout << ")\n";
}

void Mensch::test_geschlecht( void ) {
   if( geschlecht == FRAU ) {
      cout << "weiblich";
   }
   else {
      cout << "männlich";
   }
}

void Mensch::sehen( const char* objekt ) {
   cout << name << " sieht " << objekt << '\n';
}

void Mensch::hoeren( const char* geraeusch ) {
   cout << name << " hört " << geraeusch << '\n';
}

void Mensch::riechen( const char* geruch ) {
   cout << name << " riecht " << geruch << '\n';
}
```

```
void Mensch::schmecken( const char* geschmack ) {
   cout << name << " schmeckt " << geschmack << '\n';
}

void Mensch::tasten( const char* objekt ) {
   cout << name << " nimmt " << objekt << '\n';
}
```

Zum Schluss können Sie das eigentliche Hauptprogramm in eine Quelldatei schreiben. Oft wird, wie in diesem Beispiel auch, das Hauptprogramm einfach als *main.cpp* bezeichnet:

```
// main.cpp
#include "mensch.h"

int main(void) {
   // Mehrere Objekte deklarieren
   Mensch mann, frau, person, dummy;
   // Ein Objektzeiger
   Mensch* Menschptr;

   // Zugriffsmethoden aufrufen
   mann.erzeuge( "Adam", 18, Mensch::MANN );
   frau.erzeuge( "Eva" , 18, Mensch::FRAU );

   // einige Aktionen (Klassenmethoden) aufrufen
   mann.sehen( "Eva" );
   frau.sehen( "Apfel" );
   frau.tasten( "Apfel" );
   mann.hoeren( "Warnung von Gott" );
   frau.hoeren( "nichts" );
   frau.riechen( "Apfel" );
   frau.schmecken( "Apfel" );
   mann.hoeren( "Schlange" );

   // So gehts auch - dank Funktionsüberladung
   person.erzeuge( );

   // Oder indirekt mit einem Zeiger über den PfeilOperator
   Menschptr = &dummy;
   Menschptr->erzeuge( "Jürgen Wolf", 30, Mensch::MANN );

   // Ausgabe aller erzeugten Menschen
   cout << "\nMeine erzeugten Menschen bisher : \n";
   mann.print();
```

```
    frau.print();
    person.print();
    Menschptr->print();
    return 0;
}
```

Voraussetzung dafür, dass dieses Beispiel auch funktioniert, ist, dass die Header-Datei *mensch.h* im selben Verzeichnis wie die anderen beiden Quellcodes liegt (ansonsten müssen Sie die Inkludierung dieser Datei mit dem Pfad anpassen). Des Weiteren müssen Sie selbstverständlich beim Übersetzen des Quellcodes auch die Datei *mensch.cpp* mit angeben oder, bei einer GUI, im Projekt mit aufnehmen.

Hinweis **[«]**

Wie Sie dies in die Praxis übersetzen können, finden Sie auf der Buch-CD. Dort gibt es sowohl Anleitungen für Kommandozeilen-Compiler als auch für gängige Entwicklungsumgebungen.

Dadurch haben Sie praktisch die Definition der Klassen so getrennt, dass ihre Wiederverwendbarkeit erheblich erleichtert wird. Wenn Sie Ihre Klasse weitergeben wollen, aber den Quellcode nicht freigeben wollen/dürfen, so reicht es auch aus, nur die Header-Datei(en) und die übersetzte Objektdatei *(.obj/.o)* weiterzugeben. Der Anwender muss dann nur noch die Header-Datei einbinden und die entsprechende Objektdatei *(.obj/.o)* seinem Projekt hinzufügen. Beachten Sie allerdings, dass die Objektdatei plattformspezifisch ist.

Hinweis **[«]**

Um die Übersichtlichkeit in diesem Buch zu wahren und nicht Platz für seitenlangen Code zu verschwenden, wird die Klasse `Mensch` – solange diese noch verwendet wird – in die gezeigten drei einzelnen Dateien (*main.cpp*; *mensch.cpp* und *mensch.h*) aufgeteilt. Falls Sie den Überblick verlieren, finden Sie zum jeweiligen Kapitel auch das komplette Listing (mit allen Dateien) auf der Buch-CD.

4.2.8 Konstruktoren

Bisher haben Sie noch nichts über die »sicherere« Initialisierung von Variablen erfahren, womit sich C++ und die OOP immer rühmen. Ein Objekt vom Typ `Mensch` besitzt zum Beispiel so lange keinen gültigen Wert, bis die Klassenmethode `erzeuge()` aufgerufen wird. Diese trifft den Sachverhalt schon recht gut, aber Klassen bieten hierfür einen speziellen Mechanismus an, der dafür verantwortlich ist, dass ein Objekt bei der Definition automatisch mit einer Initialisierungsfunktion mit Werten für die Eigenschaften belegt wird. Einfach ausgedrückt: Es werden Standard-Initialisierungen vorgegeben, falls eine Methode mit

fehlenden Parametern aufgerufen wurde, und zusätzlich lassen sich noch verschiedene Möglichkeiten einzeln behandeln. Damit ist garantiert, dass Sie immer mit gültigen Werten arbeiten. Die Rede ist von den Konstruktoren, die im Grunde den Methoden einer Klasse recht ähnlich sind.

Doch in zwei Dingen unterscheiden sich Konstruktoren von Klassenmethoden:

- Der Name des Konstruktors ist derselbe wie der Name der Klasse.
- Der Konstruktor besitzt keinen Rückgabewert.

Konstruktoren deklarieren

Damit der Konstruktor auch zur Erzeugung von Objekten von außen zur Verfügung steht, erfolgt die Deklaration gewöhnlich im `public`-Bereich der Klasse. Hierzu müssen Sie zunächst die Deklaration der Konstruktoren in der Klasse `Mensch` vornehmen. Die einzelnen Konstruktoren (vier im Beispiel) unterscheiden sich durch ihre Parameter.

```
// mensch.h
#include <iostream>
#ifndef _MENSCH_H_
#define _MENSCH_H_
using namespace std;

class Mensch {
private:
   // Eigenschaften der Klasse Mensch
   char name[30];
   unsigned int alter;
   bool geschlecht; //0 = männlich; 1 = weiblich
   void test_geschlecht( void );

public:
   // Deklaration der Konstruktoren
   Mensch( const char*, unsigned int, bool );
   Mensch( const char*, unsigned int );
   Mensch( const char* );
   Mensch( );

   enum Geschlecht { MANN, FRAU };
   // Fähigkeiten (Methoden) der Klasse Mensch
   void sehen( const char* objekt );
   void hoeren( const char* geraeusch );
   void riechen( const char* geruch );
   void schmecken( const char* geschmack );
```

```
   void tasten( const char* objekt );
   // Einen Menschen mit allen Daten erzeugen ggf. Überladen
   void erzeuge( const char* n="Unbekannt",
                 unsigned int a=0,
                 bool g=FRAU );
   void print( void );
};
#endif
```

Konstruktoren definieren

Nachdem Sie die Deklaration der Konstruktoren in der Header-Datei geschrieben haben, müssen Sie diese, wie die Klassenmethoden auch, definieren. Die Definition ist ebenfalls den Klassenmethoden ähnlich, nur dass hierbei zweimal der Klassenname verwendet wird, gefolgt von den Parametern:

```
Klassenname::Klassenname( Parameter ) {
}
```

Natürlich können die Konstruktoren überladen werden. Daher müssen sie sich (sollten es mehrere sein) durch ihre Parameter (genauer Signatur) unterscheiden. Dadurch lassen sich die Objekte auf viele verschiedene Arten initialisieren.

In der Praxis sieht die Definition der Konstruktoren, bezogen auf die Klasse `Mensch`, wie folgt aus:

```
// mensch.cpp
#include <iostream>
#include <cstring>
#include "mensch.h"
using namespace std;

// Definition der Konstruktoren - Anfang

Mensch::Mensch( const char* n, unsigned int a, bool g ) {
   strncpy( name, n, sizeof(name)-1 );
   name[sizeof(name)] = '\0';
   alter = a;
   geschlecht = g;
}

Mensch::Mensch( const char* n, unsigned int a )  {
   strncpy( name, n, sizeof(name)-1 );
   name[sizeof(name)] = '\0';
   alter = a;
   if( a % 2 ) { // Gerade oder ungerade Zahl
```

```
      geschlecht = FRAU;
   }
   else {
      geschlecht = MANN;
   }
}


Mensch::Mensch( const char* n ) {
   strncpy( name, n, sizeof(name)-1 );
   name[sizeof(name)] = '\0';
   // Neu geboren ... :-)
   alter = 0;
   // mehr Frauen braucht das Land ;-)
   geschlecht = FRAU;
}

// Der Standardkonstruktor
Mensch::Mensch( ) {
   // Rabenvater oder Rabenmutter oder Tragödie ?
   strncpy( name, "Unbekannt", sizeof(name)-1 );
   name[sizeof(name)] = '\0';
   // Neu geboren ... :-)
   alter = 0;
   // als Ausgleich wieder ein Mann ;-)
   geschlecht = MANN;
}

// Definition der Konstruktoren - Ende

void Mensch::erzeuge(const char* n,unsigned int a,bool g) {
   // Hier könnten/sollten die übergebenen Parameter
   // überprüft werden, bevor diese verwendet werden
   strncpy( name, n, sizeof(name)-1 );
   name[sizeof(name)] = '\0';
   alter = a;
   geschlecht = g;
}

void Mensch::print ( void ) {
   cout << name << " " << alter << " Jahre (";
   test_geschlecht();
   cout << ")\n";
}
```

```
void Mensch::test_geschlecht( void ) {
   if( geschlecht == FRAU ) {
      cout << "weiblich";
   }
   else {
      cout << "männlich";
   }
}
void Mensch::sehen( const char* objekt ) {
   cout << name << " sieht " << objekt << '\n';
}

void Mensch::hoeren( const char* geraeusch ) {
   cout << name << " hört " << geraeusch << '\n';
}

void Mensch::riechen( const char* geruch ) {
   cout << name << " riecht " << geruch << '\n';
}

void Mensch::schmecken( const char* geschmack ) {
   cout << name << " schmeckt " << geschmack << '\n';
}

void Mensch::tasten( const char* objekt ) {
   cout << name << " nimmt " << objekt << '\n';
}
```

Die Definition von Konstruktoren überprüft die übergebenen Argumente auf ihre Gültigkeit und kopiert die entsprechenden Eigenschaften in das Objekt. Wenn eine Eigenschaft eines Objekts nicht initialisiert wurde, so sollte man diese mit einem Standardwert (Nullwert) initialisieren.

Neben den Initialisierungen von Variablen werden Konstruktoren auch für andere Dinge verwendet. Gängige und häufig gebrauchte Anwendungen sind das Reservieren von Speicher, das Öffnen von Dateien oder das Vorbereiten von Daten usw.

Konstruktoren aufrufen

Da ein Konstruktor keinen Rückgabewert besitzt, können Sie diesen nicht wie eine normale »Klassenmethode« aufrufen. Daher kann der Aufruf nur implizit bei der Definition des Objekts erfolgen. Die Syntax lautet daher:

```
Klassenname Objekt ( Parameterliste );
```

Als Nächstes sucht der Compiler nach einem passenden Konstruktor mit der passenden Signatur. Gibt es einen, erzeugt der Compiler ein Objekt mit den entsprechenden Argumenten, die bei der Definition mit übergeben wurden. Gibt es keinen passenden Konstruktor, bricht der Compiler die Übersetzung ab und gibt eine Fehlermeldung aus.

Im Beispiel der Klasse `Mensch` wurden vier Konstruktoren verwendet. Somit stehen Ihnen folgende Möglichkeiten zur Verfügung, ein neues Objekt mit definierten Eigenschaften anzulegen:

```
// main.cpp
#include "mensch.h"

int main(void) {
   Mensch mann1("Adam", 18, Mensch::MANN );
   Mensch frau1("Eva", 18, Mensch::FRAU );
   Mensch person1("Jürgen", 30 );
   Mensch person2("Fatma" );
   // Verwendet Standardkonstruktor
   Mensch person3;
   // Auch möglich ...
   Mensch person4 = "Mathilda";
   // Natürlich geht es auch indirekt ...
   // Verwendet Standardkonstruktor
   Mensch person5;
   Mensch* MenschPtr = &person5;

   mann1.print();
   frau1.print();
   person1.print();
   person2.print();
   // Ohne Konstruktor würde hier ein
   // undefiniertes Verhalten vorliegen
   person3.print();
   person4.print();
   // Indirekter Zugriff auf person5
   MenschPtr->print();
   return 0;
}
```

Die Ausgabe des Programms:

```
Adam 18 Jahre (männlich)
Eva 18 Jahre (weiblich)
Jürgen 30 Jahre (männlich)
Fatma 0 Jahre (weiblich)
```

```
Unbekannt 0 Jahre (männlich)
Mathilda 0 Jahre (weiblich)
Unbekannt 0 Jahre (männlich)
```

Natürlich decken diese Konstruktoren noch lange nicht alles ab. Wenn Sie zum Beispiel ein Objekt mit der Eigenschaft `MANN` wie folgt erzeugen wollen

```
Mensch person6( Mensch::MANN );
```

bekommen Sie eine Fehlermeldung zurück, da es keinen passenden Konstruktor dazu gibt. Aber so etwas ließe sich problemlos nachrüsten:

```
// Deklaration in mensch.h
Mensch( bool );
...
// Definition in mensch.cpp
Mensch::Mensch( bool g ) {
   // Rabenvater oder Rabenmutter oder Tragödie ?
   strncpy( name, "Unbekannt", sizeof(name)-1 );
   name[sizeof(name)] = '\0';
   // Neu geboren ... :-)
   alter = 0;
   // Geschlecht bestimmt der Anwender ... :-)
   geschlecht = g;
}
```

Sicherlich ist Ihnen auch folgendes Objekt ins Auge gestochen:

```
Mensch person4 = "Mathilda";
```

Dies können Sie verwenden, falls Sie ein Objekt mit nur einer Eigenschaft initialisieren und natürlich ein entsprechender Konstruktor vorhanden ist. Das bedeutet auch, dass Sie ein Objekt folgendermaßen definieren können

```
Mensch person6 = Mensch::MANN;
```

wenn Sie den beschriebenen Konstruktor

```
Mensch( bool );
```

deklariert und definiert haben.

Hinweis **[«]**

Die Verwendung mit der Initialisierung zwischen den Klammern ist übrigens nicht auf Klassen beschränkt, sondern lässt sich theoretisch auch auf die Basisdatentypen anwenden: `int iwert(5)` **statt** `int iwert = 5`.

Standardkonstruktor

Konstruktoren ohne Parameter heißen Standardkonstruktoren (Default-Konstruktoren). Diese werden immer dann aufgerufen, wenn bei der Definition eines Objekts keine weiteren Initialisierungswerte angegeben wurden:

```
// Verwendet Standardkonstruktor
Klassenname Objekt;
```

In der Praxis schreibt man diesen Standardkonstruktor gewöhnlich so, dass alle Eigenschaften einer Klasse mit Standardwerten versehen werden. Im Beispiel zuvor sah die Definition des Standardkonstruktors wie folgt aus:

```
// Deklaration des Standardkonstruktors in mensch.h
Mensch( );
...
// Definition des Standardkonstruktors in mensch.cpp
Mensch::Mensch( ) {
   // Rabenvater oder Rabenmutter oder Tragödie ?
   strncpy( name, "Unbekannt", sizeof(name)-1 );
   name[sizeof(name)] = '\0';
   // Neu geboren ... :-)
   alter = 0;
   // als Ausgleich wieder ein Mann ;-)
   geschlecht = MANN;
}
```

Im Hauptprogramm wird dieser Standardkonstruktor zum Beispiel mit

```
// Verwendet Standardkonstruktor
Mensch person3;
```

verwendet.

Sofern Sie keinen Standardkonstruktor verwenden (was man in der Praxis allerdings immer machen sollte), stellt Ihnen der Compiler automatisch einen solchen zur Verfügung. Dieser Standardkonstruktor (ebenfalls von außen erreichbar als `public`) beschreibt allerdings nicht die Eigenschaften einer Klasse mit Werten.

Beachten Sie auch, dass Ihnen der Compiler keinen Standardkonstruktor mehr zur Verfügung stellt, sobald Sie mindestens einen Konstruktor verwenden, und Sie sich selbst darum kümmern müssen. Würden Sie den Standardkonstruktor im Beispiel entfernen, so würde folgende Objektdefinition zu einem Compiler-Fehler führen:

```
Mensch person3;
```

Dies hätte allerdings wieder den Vorteil, dass Sie bei der Definition eines Objekts eine Initialisierungsliste mit angeben müssen.

4.2.9 Destruktoren

Der Destruktor wird genauso wie der Konstruktor automatisch aufgerufen – mit dem Unterschied, dass dies nicht bei der Definition geschieht, sondern beim »Zerstören« des Objekts. Der Destruktor wird gewöhnlich für Aufräumarbeiten wie das Freigeben von dynamisch reserviertem Speicherplatz verwendet. Man spricht auch vom Abbauen eines Objekts.

Destruktor deklarieren

Der Destruktor besteht, wie auch der Konstruktor, nur aus dem Klassennamen – allerdings mit einem vorangestellten ~ (Tilde-Zeichen):

```
// Deklaration eines Destruktors
~Klassenname( );
```

Also gilt auch für den Destruktor, dass dieser keinen Rückgabewert enthält, und außerdem besitzt er niemals einen Parameter – im Gegensatz zum Konstruktor. Darüber hinaus muss der Destruktor immer im `public`-Bereich einer Klasse stehen.

Destruktor definieren

Sofern Sie keinen Destruktor definieren, erzeugt der Compiler eine Standardversion davon (im `public`-Bereich). Dieser Destruktor zerstört die Eigenschaften eines Objekts in umgekehrter Reihenfolge (da Stack) der Erzeugung. Dies gilt auch, wenn die Eigenschaft selbst ein Objekt ist – nur wird dann der Destruktor (implizit oder, falls vorhanden, explizit) verwendet.

Wenn Sie einen Destruktor selbst explizit definieren, wird dieser ähnlich wie der Konstruktor definiert. Die Syntax hierzu lautet:

```
// Definition eines Destruktors
Klassenname::~Klassenname( ) {
   // Anweisungen
}
```

In der Klasse `Mensch` wird die Deklaration des Destruktors im `public`-Bereich vorgenommen. Hierzu die Header-Datei *mensch.h*, ein wenig gekürzt:

```
// mensch.h
#include <iostream>
#ifndef _MENSCH_H_
#define _MENSCH_H_
using namespace std;

class Mensch {
private:
```

```
   // Eigenschaften der Klasse Mensch
   char name[30];
   unsigned int alter;
   bool geschlecht; //0 = männlich; 1 = weiblich
   void test_geschlecht( void );

public:
   // Deklaration der Konstruktoren
   Mensch( const char*, unsigned int, bool );
   Mensch( const char*, unsigned int );
   Mensch( const char* );
   Mensch( bool );
   Mensch( );
   // Deklaration des Destruktors
   ~Mensch( );

   enum Geschlecht { MANN, FRAU };
   // Fähigkeiten (Methoden) der Klasse Mensch

   // Deklarationen der Methoden sehen(), hoeren(),
   // riechen(), schmecken(), tasten() hierher schreiben

   void print( void );
};
#endif
```

Die Definition des Destruktors in der Quelldatei *mensch.cpp* kann demnach wie folgt aussehen (auch gekürzt):

```
// mensch.cpp
#include <iostream>
#include <cstring>
#include "mensch.h"
using namespace std;

// Definition der Konstruktoren - Anfang
...
// Hier kommen die Konstruktoren hin
...
// Definition der Konstruktoren - Ende

// Definition des Destruktors
Mensch::~Mensch( ) {
   cout << name << "ist von uns gegangen\n";
}
```

```
// Hier kommen die restlichen Methoden hin
...
```

Destruktor aufrufen (implizit)

Hierzu nun die `main`-Funktion, die den Destruktor in der Praxis demonstrieren soll:

```
// main.cpp
#include "mensch.h"

int main(void) {
   Mensch person1( "Adam", 18, Mensch::MANN );
   {
      Mensch person2( "Justus", 30, Mensch::MANN );
      static Mensch person3 ("Johanna", 100, Mensch::FRAU );
   }
   cout << "Der Rest geht nach dem Programmende\n";
   return 0;
}
```

Das Programm bei der Ausführung:

```
Justus ist von uns gegangen
Der Rest geht nach dem Programmende
```

Der Destruktor kann nicht explizit aufgerufen werden, sondern wird immer implizit verwendet, wenn sich der Gültigkeitsbereich des Objekts beendet. Für Objekte gilt dasselbe, was in Kapitel 3, »Gültigkeitsbereiche, spezielle Deklarationen und Typumwandlungen«, für den Gültigkeitsbereich von anderen Typen beschrieben wurde. Ist ein Objekt

- lokal deklariert und gehört nicht zur Speicherklasse `static`, wird dieses am Ende des entsprechenden Anweisungsblocks zerstört.
- `global` oder `static` deklariert, wird es gewöhnlich am Ende des Programms zerstört (weshalb im Beispiel auch keine Ausgabe mehr auf dem Bildschirm erfolgte).

4.3 Mehr zu den Klassenmethoden (Klassenfunktionen)

In den nächsten Abschnitten soll auf typische Themen der Klassenmethoden eingegangen werden – was natürlich voraussetzt, dass Sie den Abschnitt zu den Klassen verstanden haben.

4.3.1 Inline-Methoden (explizit und implizit)

Wie Sie bereits bei den Funktionen erfahren haben, stellt deren Aufruf keinen unerheblichen Aufwand da – Stichwort: Stack(-Frame). Dasselbe gilt auch in Bezug auf die Klassenmethoden, die im Grunde auch nur Funktionen (für eine Klasse) sind. Im Gegensatz zu »gewöhnlichen« Funktionen ist der Aufwand, der für den Aufruf von Klassenmethoden betrieben werden muss, noch erheblich höher.

Auch hier steht Ihnen wie bei den Funktionen die Möglichkeit zur Verfügung, eine Klassenmethode als `inline` zu definieren (siehe Abschnitt 1.10.6, »Inline-Funktionen«). Damit können Sie die Vorteile der Klassen wieder nutzen, ohne dass diese zu einem schlechten Laufzeitverhalten beitragen.

Zur Verwendung von `inline` haben Sie bei den Klassen zwei Möglichkeiten, die explizite und die implizite.

Inline implizit

Wenn kleinere Funktionen gleich innerhalb der Klasse definiert werden, werden sie automatisch zu `inline`-Klassenmethoden, auch wenn das Schlüsselwort `inline` nicht mit angegeben wird. Natürlich können Sie diese Methoden trotzdem zur besseren Lesbarkeit extra mit `inline` definieren. Zum Beispiel die Klasse `Mensch`:

```
// mensch.h
#include <iostream>
#ifndef _MENSCH_H_
#define _MENSCH_H_
using namespace std;

class Mensch {
private:
   // Eigenschaften der Klasse Mensch
   char name[30];
   unsigned int alter;
   bool geschlecht; //0 = männlich; 1 = weiblich
   void test_geschlecht( void );

public:
   // Deklaration der Konstruktoren
   Mensch( const char*, unsigned int, bool );
   Mensch( const char*, unsigned int );
   Mensch( const char* );
   Mensch( bool );
```

```
   Mensch( );
   // Deklaration der Destruktoren
   ~Mensch( );

   enum Geschlecht { MANN, FRAU };
   // Fähigkeiten (Methoden) der Klasse Mensch
   // Die folgenden Methoden implizit als inline definieren
   void sehen( const char* objekt ) {
      cout << name << " sieht " << objekt << '\n';
   }

   void hoeren( const char* geraeusch ) {
      cout << name << " hört " << geraeusch << '\n';
   }

   void riechen( const char* geruch ) {
      cout << name << " riecht " << geruch << '\n';
   }

   void schmecken( const char* geschmack ) {
      cout << name << " schmeckt " << geschmack << '\n';
   }

   void tasten( const char* objekt ) {
      cout << name << " nimmt " << objekt << '\n';
   }

   // Einen Menschen mit allen Daten erzeugen ggf. Überladen
   void erzeuge( const char* n="Unbekannt",
                 unsigned int a=0,
                 bool g=FRAU );
   void print( void );
};
#endif
```

In diesem Beispiel gelten die Klassenmethoden `sehen()`, `hoeren()`, `riechen()`, `schmecken()` und `tasten()` implizit als `inline`.

Inline explizit

Klassenmethoden können natürlich auch explizit als `inline` definiert werden, was durchaus die gängigere Methode darstellt. Hierbei geht man genauso vor wie schon bei den `inline`-Funktionen und setzt bei der Definition vor der Klassenmethode das Schlüsselwort `inline`.

Auf unser Beispiel der Klasse `Mensch` bezogen, müssten Sie hierzu in der Datei *mensch.cpp* vor den entsprechenden Methoden das Schlüsselwort `inline` setzen (natürlich setzt dies voraus, dass Sie in der Header-Datei *mensch.h* noch keine Definition, wie mit »inline implizit« gesehen, vorgenommen haben). Hier das Beispiel dazu:

```
// mensch.cpp
#include <iostream>
#include <cstring>
#include "mensch.h"
using namespace std;

// ... alles wie gehabt - gekürzt
...

inline void Mensch::sehen( const char* objekt ) {
   cout << name << " sieht " << objekt << '\n';
}

inline void Mensch::hoeren( const char* geraeusch ) {
   cout << name << " hört " << geraeusch << '\n';
}

inline void Mensch::riechen( const char* geruch ) {
   cout << name << " riecht " << geruch << '\n';
}

inline void Mensch::schmecken( const char* geschmack ) {
   cout << name << " schmeckt " << geschmack << '\n';
}

inline void Mensch::tasten( const char* objekt ) {
   cout << name << " nimmt " << objekt << '\n';
}
```

Inline-Konstruktoren und -Destruktoren

Natürlich können Sie, wie auch die Klassenmethoden, die Konstruktoren und Destruktoren als `inline` definieren. Und das sowohl explizit als auch implizit, nur muss auf die »Merkmale« von Konstruktoren und Destruktoren geachtet werden.

Hinweis [«]

Natürlich gilt auch bei der Verwendung von `inline` bei den Klassenmethoden bzw. den Konstruktoren/Destruktoren dasselbe wie bei den Funktionen. Mit dem Schlüsselwort `inline` (wenn es explizit erfolgt) oder mit der beabsichtigten impliziten inline-Definition fordern Sie den Compiler nur auf, diese, wenn möglich, als `inline` einzubauen. Es handelt sich also um keine Vorschrift. Aber in der Praxis sind die Compiler bereits selbst so »intelligent« und versuchen, eigenständig Klassenmethoden bzw. Konstruktoren/Destruktoren als `inline` einzubauen, besonders wenn eine Methode in der gleichen Datei verwendet wird.

4.3.2 Zugriffsmethoden

Das Thema der Zugriffsmethoden wurde bereits kurz erwähnt. Diese dienen dazu, auf die `private`-Eigenschaften (Daten) einer Klasse zuzugreifen. Man kann diese »Datenkapselung« zwar unterlaufen, wenn man die Eigenschaften als `public` deklariert, aber dies ist wohl eher nicht im Sinne der OOP.

Daher werden in der Praxis die Zugriffsmethoden verwendet, die im Prinzip auch nur einfache Methoden sind, um auf die Eigenschaften einer Klasse zugreifen zu können. Im Beispiel der Klasse `Mensch` wurde die Methode `erzeuge()` verwendet, um die Eigenschaften der Klasse mit einem Wert zu initialisieren. Was hier allerdings noch fehlt, sind Methoden, um einzelne Eigenschaften dieser Klasse abzufragen bzw. zu (über)schreiben.

In der Praxis wird hierfür oft die Syntax

```
get_eigenschaft();
```

verwendet, um Daten zu holen (`get_`) und

```
set_eigenschaft();
```

zum (Über-)Schreiben bzw. Setzen (`set_`) von Daten. Im Beispiel der Klasse `Mensch` könnten Sie somit das Alter einer Person mit

```
Mensch person;
...
person.get_alter();
```

ermitteln und mit

```
person.set_alter( 23 );
```

(über-)schreiben bzw. setzen.

Bezogen auf die Klasse `Mensch`, müssen Sie zunächst die Header-Datei um die Deklarationen der neuen Zugriffsmethoden erweitern:

```
// mensch.h
#include <iostream>

#ifndef _MENSCH_H_
#define _MENSCH_H_
using namespace std;

class Mensch {
private:
   // Eigenschaften der Klasse Mensch
   char name[30];
   unsigned int alter;
   bool geschlecht; //0 = männlich; 1 = weiblich
   void test_geschlecht( void );

public:
   // Deklaration der Konstruktoren
   Mensch( const char*, unsigned int, bool );
   Mensch( const char*, unsigned int );
   Mensch( const char* );
   Mensch( bool );
   Mensch( );
   // Deklaration der Destruktoren
   ~Mensch( );

   enum Geschlecht { MANN, FRAU };
     // Fähigkeiten (Methoden) der Klasse Mensch
   void sehen( const char* objekt );
   void hoeren( const char* geraeusch );
   void riechen( const char* geruch );
   void schmecken( const char* geschmack );
   void tasten( const char* objekt );
   // Einen Menschen mit allen Daten erzeugen ggf. Überladen
   void erzeuge( const char* n="Unbekannt",
                 unsigned int a=0,
                 bool g=FRAU );
   // Zugriffsmethoden zum Abfragen der Eigenschaften
   const char* get_name( void );
   unsigned int get_alter( void );
   bool get_geschlecht( void );
   // Zugriffsmethoden zum Setzen der Eigenschaften
   void set_name( const char* n );
   void set_alter( unsigned int a );
   void set_geschlecht( bool g );
   // Ausgabe aller Eigenschaften
```

```
   void print( void );
};
#endif
```

Die einzelnen Zugriffsmethoden werden dann in der Datei *mensch.cpp* wie folgt definiert (wie immer gekürzt):

```
// mensch.cpp
#include <iostream>
#include <cstring>
#include "mensch.h"
using namespace std;

// ... hierhin alles wie gehabt
...

// Zugriffsmethoden zum Abfragen der Eigenschaften

const char* Mensch::get_name( void ) {
   return name;
}

unsigned int Mensch::get_alter( void ) {
   return alter;
}

bool Mensch::get_geschlecht( void ) {
   return geschlecht;
}

// Zugriffsmethoden zum Setzen der Eigenschaften

void Mensch::set_name( const char* n ) {
   strncpy( name, n, sizeof(name)-1 );
   name[sizeof(name)] = '\0';
}

void Mensch::set_alter( unsigned int a ) {
   alter = a;
}

void Mensch::set_geschlecht( bool g ) {
   geschlecht = g;
}
```

Und wieder eine *main*-Datei, die die hier deklarierten und definierten Zugriffsmethoden in der Praxis bei der Ausführung zeigen soll:

```
// main.cpp
#include "mensch.h"

int main(void) {
   Mensch person1( "Adam", 18, Mensch::MANN );
   Mensch person2;
   person1.print();
   person2.print();

   person1.set_alter( 20 );
   person2.set_name("Eva");
   person2.set_alter( 19 );
   person2.set_geschlecht( Mensch::FRAU );
   person1.print();
   person2.print();

   if( person1.get_alter() > person2.get_alter() ) {
      cout << person1.get_name() << " ist älter als "
           << person2.get_name() << "\n";
   }
   else {
      cout << person2.get_name() << " ist älter als "
           << person1.get_name() << '\n';
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Adam 18 Jahre (männlich)
Unbekannt 0 Jahre (männlich)
Adam 20 Jahre (männlich)
Eva 19 Jahre (weiblich)
Adam ist älter als Eva
```

Zunächst erscheinen einem die Zugriffsmethoden als enormer Mehraufwand, und der Quellcode wird dadurch auch nicht »kürzer«. Aber in der Praxis überwiegen die Vorteile eindeutig.

Zum einen können Sie jetzt mit solchen Zugriffsmethoden fehlerhafte Zugriffe auf Variablen von vornherein ausschließen. Damit können Sie sicher sein, dass Objekte stets mit gültigen (definierten) Werten arbeiten.

Und mit Zugriffsmethoden werden die implementierungsabhängigen Details einer Klasse versteckt. Man kann jederzeit die Klassenmethoden verändern (etwa die Darstellung von Daten), ohne dass eine Zeile des Anwendungsprogramms selbst geändert werden muss – dies gilt natürlich nur, solange die öffentlichen Schnittstellen (die ja die Methoden sind) unverändert bleiben.

4.3.3 Read-only-Methoden

Klassenmethoden, die nur lesend auf die Eigenschaften zugreifen, werden in der Praxis auch speziell gekennzeichnet. Damit ist es möglich, dass diese mit `const`-Objekten aufgerufen werden. Eine Klassenmethode wird als Read-only deklariert und definiert, indem man an den Funktionskopf das Schlüsselwort `const` anhängt:

```
// Deklaration in der Klasse
typ methode( parameter ) const;
...
// Definition der Methode
typ methode( parameter ) const {
   // Anweisungen
}
```

Im Beispiel der Klasse `Mensch` würde man beispielsweise die Zugriffsmethoden `get_` als Read-only deklarieren und definieren. Hierzu müssen Sie nur an den entsprechenden Stellen das Schlüsselwort `const` anhängen. In der Header-Datei *mensch.h* beträfe dies folgende drei Methoden:

```
// mensch.h
#include <iostream>
#ifndef _MENSCH_H_
#define _MENSCH_H_
using namespace std;

class Mensch {
private:
   // Eigenschaften der Klasse Mensch
   char name[30];
   unsigned int alter;
   // 0 = männlich; 1 = weiblich
   bool geschlecht;
   void test_geschlecht( void );

public:
   ...
   // Alles wie gehabt
```

```
   ...
   // Zugriffsmethoden zum Abfragen der Eigenschaften
   const char* get_name( void ) const;
   unsigned int get_alter( void ) const;
   bool get_geschlecht( void ) const;
};
#endif
```

Neben der Deklaration muss eine solche Methode natürlich auch in der Definition mit `const` signiert werden. Hierzu müssen Sie in der Datei *mensch.cpp* die folgenden drei Zugriffsmethoden anpassen:

```
// mensch.cpp
#include <iostream>
#include <cstring>
#include "mensch.h"
using namespace std;
...
// Alles wie bisher
...

// Zugriffsmethoden zum Abfragen der Eigenschaften

const char* Mensch::get_name( void ) const {
   return name;
}

unsigned int Mensch::get_alter( void ) const {
   return alter;
}

bool Mensch::get_geschlecht( void ) const {
   return geschlecht;
}
...
```

Der Vorteil solcher Read-only-Methoden besteht darin, dass aus ihnen keine anderen Methoden aufgerufen werden können, die die Eigenschaften eines Objekts überschreiben. Zum Beispiel würde der Compiler folgenden Codeabschnitt bemängeln und das Programm nicht übersetzen:

```
bool Mensch::get_geschlecht( void ) const {
   // !!! Fehler - nicht erlaubt !!!
   set_geschlecht( FRAU );
   return geschlecht;
}
```

Folgende Aspekte sollten allerdings in Bezug auf Read-only-Methoden noch erwähnt werden: Zum einen können diese Methoden natürlich auch für nicht konstante Objekte aufgerufen werden, und zum anderen gehört das Schlüsselwort `const` zur Signatur einer Funktion (Methode). Dadurch ist es theoretisch auch möglich, dass Sie zwei Versionen derselben Funktion implementieren – für konstante Objekte eine Read-only-Version und für nicht konstante Objekte eine andere:

```
// ... für konstante Objekte
typ methode( parameter ) const;
// ... für nicht konstante Objekte
typ methode( parameter );
...
...
// Definition
typ methode( parameter ) const {
   // Anweisungen
}

typ methode( parameter ) {
   // Anweisungen
}
```

4.3.4 this-Zeiger

Sicherlich haben Sie sich schon gefragt, wie es möglich sein kann, dass man mit den Klassenmethoden auf die Eigenschaften eines bestimmten Objekts zugreifen kann, obwohl niemals eine explizite Angabe zum Objekt gemacht wurde. Als Beispiel dient erneut die Klasse `Mensch`. Wenn Sie folgendes Objekt ausgeben wollen

```
Mensch person;
person2.print();
```

wird hierbei ja die Klassenmethode `print()` aufgerufen:

```
void Mensch::print ( void ) {
   cout << name << " " << alter << " Jahre (";
   test_geschlecht();
   cout << ")\n";
}
```

Jetzt ist aber bei der Klassenmethode keine Angabe in Sicht, mit welchem Objekt diese Methode eigentlich arbeitet. Die Antwort lautet, dass beim Aufruf einer jeden Klassenmethode als nicht sichtbares Argument die Adresse des aktuellen Objekts mit übergeben wird. Diese Adresse steht in der Klassenmethode mit dem konstanten Zeiger `this` zur Verfügung. Die Syntax der Deklaration dieses `this`-Zeigers sieht intern folgendermaßen aus:

```
Klassenname* const this = &Objekt;
```

Auf das Beispiel `Mensch person` bezogen, sieht das so aus:

```
Mensch* const this = &person;
```

Somit ist also das Objekt `person` das Objekt, mit dem die Klassenmethode aufgerufen wird.

[»] **Hinweis**

Wie Sie schon der Syntax des `this`-Zeigers entnehmen können, handelt es sich um einen konstanten Zeiger, den Sie nicht mehr verändern können. Der Zeiger verweist also immer auf das aktuelle Objekt – wobei natürlich das Objekt selbst verändert werden kann.

this-> und *this

Verwenden Sie `this` innerhalb von Klassenmethoden, erfolgt der Zugriff auf die einzelnen Elemente einer Klasse mit folgendem Ausdruck:

```
this->Element
```

In der Tat handelt es sich hierbei tatsächlich darum, was der Compiler implizit daraus machen würde, wenn Sie nur den Ausdruck

```
Element
```

verwenden würden. Dem entspricht, bezogen auf die Klasse `Mensch` und die Klassenmethode (bzw. hier Zugriffsmethode) `set_name()`, folgende Definition:

```
void Mensch::set_name( const char* n ) {
   strncpy( name, n, sizeof(name)-1 );
   name[sizeof(name)] = '\0';
}
```

Das ist dasselbe wie folgende Definition:

```
void Mensch::set_name( const char* n ) {
   strncpy( this->name, n, sizeof(this->name)-1 );
   this->name[sizeof(this->name)] = '\0';
}
```

Wenn der Compiler dies per Standard selbst so vornimmt, stellt sich natürlich die Frage nach dem Sinn von `this->`. In der Praxis kann man einen expliziten `this`-Zeiger verwenden, um die Bezeichner von lokalen Variablen einer Klassenmethode von den Eigenschaften (Klassenvariablen) mit gleichem Namen zu unterscheiden (wenn ein gleicher Name verwendet werden sollte).

Bezogen auf die Zugriffsmethode `set_name()`, würde dies in der Praxis so aussehen:

```
void Mensch::set_name( const char* name ) {
   strncpy( this->name, name, sizeof(this->name)-1 );
   this->name[sizeof(this->name)] = '\0';
}
```

Hieran erkennt man gleich die lokale Variable (`name`) und die Klassenvariable (Eigenschaft, `this->name`). Natürlich lässt sich der `this`-Zeiger auch bei den Konstruktoren und Destruktoren einsetzen.

Die zweite Verwendung des `this`-Zeigers lautet `*this`. Über diesen Zeiger können Sie auf ein Objekt komplett zugreifen (als Ganzes). Dies wird häufig verwendet, wenn aus einer Klassenmethode ein Objekt als Kopie oder Referenz zurückgegeben werden soll (`return *this`). Auf diese Art der Verwendung von `*this` wird auf den kommenden Seiten noch näher eingegangen.

4.4 Verwenden von Objekten

Dieser Abschnitt wird Ihnen sicherlich wie eine Wiederholung des Umgangs mit den verschiedenen Datentypen vorkommen – nur bezieht sich das Ganze nun auf die Objekte.

4.4.1 Read-only-Objekte

Wie jeden anderen Typ können Sie auch Objekte mit `const` deklarieren, damit dieses Objekt nur noch lesbar ist:

```
const Mensch person1( "Adam", 18, Mensch::MANN );
```

Aber seien Sie gewarnt: Der Compiler kann nämlich ohne zusätzliche Informationen nicht wissen, ob eine Klassenmethode nur lesen oder auch schreiben will. Dies bedeutet in der Praxis, dass neben den `set_`-(Zugriffs-)Klassenmethoden zum Beschreiben der Eigenschaften auch die `get_`-(Zugriffs-)Klassenmethoden nicht aufgerufen werden können.

4.4.2 Objekte als Funktionsargumente

Wie bei den bereits kennengelernten Typen stehen Ihnen bei der Übergabe von Objekten als Argument an Funktionen drei Möglichkeiten zur Verfügung, und zwar per `call by value`, `call by reference` und `Referenzen`.

Call by value

Der Aufwand bei der Übergabe *by value* an eine Funktion kann enorm sein, da hierbei eine komplette Kopie des Objekts erzeugt wird, womit die Funktion anschließend arbeitet. Das bedeutet, bei der Übergabe *by value* wird der Konstruktor aufgerufen und ein neues Objekt für die Kopie erzeugt. Beim Beenden der Funktion wird das Objekt wieder zerstört – wobei hier auch der Destruktor aufgerufen werden muss. Ein ziemlicher Aufwand eben.

Als Beispiel können Sie ja am Ende der Header-Datei *mensch.h* folgende globale Funktionsdeklaration einfügen:

```
// mensch.h
#include <iostream>
#ifndef _MENSCH_H
#define _MENSCH_H
using namespace std;

class Mensch {
    // Hier die Eigenschaften und Methoden wie gehabt
    ...
};

// Globale Funktion
extern int vergleiche_alter(
   const Mensch p1, const Mensch p2 );

#endif
```

Die Definition der Funktion sollten Sie sauber von der Deklaration trennen und diese am besten im Quellcode `mensch.cpp` wie folgt vornehmen:

```
// mensch.cpp
#include <iostream>
#include <cstring>
#include "mensch.h"
using namespace std;

// Definition von Konstruktoren, Destruktor
// wie gehabt und Zugriffsmethoden
...
...
// Am Ende die Definition der gloablen Funktion
int vergleiche_alter( const Mensch p1, const Mensch p2 ) {
   return p1.get_alter() - p2.get_alter();
}
```

Anwenden können Sie diese Funktion dann beispielsweise wie folgt:

```
// main.cpp
#include "mensch.h"

int main(void) {
   Mensch person1( "Adam", 18, Mensch::MANN );
   Mensch person2( "Eva", 19, Mensch::FRAU );
   int ret;
   ret = vergleiche_alter( person1, person2 );
   if( ret < 0 ) {
      cout << person1.get_name() << " ist älter als "
           << person2.get_name() << '\n';
   }
   else if( ret == 0 ) {
      cout << "Beide Personen sind gleich alt\n";
   }
   else {
      cout << person2.get_name() << " ist älter als "
           << person1.get_name() << '\n';
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Adam ist von uns gegangen
Eva ist von uns gegangen
Adam ist älter als Eva
```

Bei der Ausführung des Programms können Sie erkennen, dass hier der Destruktor aktiv war – und zwar in der Funktion `vergleiche_alter()`. Beim Aufruf dieser Funktion werden hierbei zwei Objekte der Klasse `Mensch` an die beiden Parameter kopiert, also vom Konstruktor neu erzeugt. Beim Verlassen wiederum werden diese zerstört – also der Destruktor aufgerufen.

Call by reference

Dass die Verwendung des *call by value*-Verfahrens eher kontraproduktiv für das Laufzeitverhalten eines Programms ist, dürfte wohl klar sein. Diesen Aufwand kann man ohne Probleme mit *call by reference* vermeiden (siehe Abschnitt 2.6.2, »call by reference – Zeiger als Funktionsparameter«). Da hier mit den (Original-) Adressen der Objekte gearbeitet wird, sollte natürlich auch klar sein, dass sich jede Veränderung einer `public`-Eigenschaft auf das tatsächliche Objekt auswirkt. Aber in der Regel sind die Eigenschaften einer Klasse fast immer `private`.

Bezogen auf das Beispiel der Klasse `Mensch` und der zuvor erstellten globalen Funktion `vergleiche_alter()`, ist bei der Funktionsdefinition nicht viel zu verändern. Lediglich die Parameter dieser Funktion müssen als Zeiger angegeben werden, und der Zugriff auf die Klassenmethoden bzw. `public`-Eigenschaften erfolgt über den Pfeiloperator (`->`):

```
int vergleiche_alter( const Mensch* p1, const Mensch* p2 ) {
   return p1->get_alter() - p2->get_alter();
}
```

Beim Aufruf der Funktion müssen Sie natürlich auch die Adressen der Objekte mit Hilfe des Adressoperators übergeben:

```
ret = vergleiche_alter( &person1, &person2 );
```

Hiermit entfällt der Aufwand, der für das Kopieren und Zerstören eines Objekts benötigt wird. Daher empfiehlt es sich immer, Objekte an Funktionen über Zeiger oder Referenztypen zu übergeben.

Referenzen auf Objekte

Natürlich können Sie auch Referenzen auf Objekte verwenden. Hierbei haben Sie dieselben Vorzüge von Zeigern und den gleichen (einfacheren) Komfort von normalen Variablen (Zugriff und Verwendung) – siehe auch Abschnitt 2.2, »Referenzen«.

Die Definition der Funktion `vergleiche_alter()` sieht mit Referenzen auf Objekte wie folgt aus:

```
int vergleiche_alter(const Mensch& p1, const Mensch& p2 ) {
   return p1.get_alter() - p2.get_alter();
}
```

Der Funktionsaufruf und die Anwendung in der Funktion selbst auf die einzelnen Zugriffsmethoden und gegebenenfalls `public`-Eigenschaften erfolgt hingegen wie bei den normalen Variablen.

Hier haben Sie außerdem durch die Deklaration von `const` (konstante Referenzen) der beiden Parameter einen Schreibschutz (Read-only) verwendet. Damit bleiben die Daten wie beim *call by value*-Verfahren geschützt.

Klassenmethoden mit Objekten als Argumente

Im Beispiel mit `vergleiche_alter()` wurde eine globale Funktion verwendet, was allerdings nicht so ganz in das OOP-Denkschema passt. In der OOP geht man immer davon aus, dass ein Objekt sich »selbst« verwaltet. Im Klartext heißt das, dass man innerhalb der Klasse eine Klassenmethode entwerfen sollte, die diese

Aufgabe löst. Natürlich ist hierbei auch die Methodenstellung anders, da Methoden ja mit den Eigenschaften (Daten) eines Objekts arbeiten:

```
Objekt1.Methode( )
```

Außerdem ist es mit den globalen Funktionen nicht möglich, auf die privaten Eigenschaften einer Klasse zuzugreifen. Daraus ergibt sich im Beispiel `vergleiche_alter()`, dass die Klassenfunktion die Eigenschaften des aktuellen Objekts mit den Eigenschaften eines anderen Objekts vergleicht, weshalb Sie also nur einen Parameter für das zweite Objekt benötigen:

```
Objekte1.Method( Objekt2 )
```

Erst damit ist es möglich, dass auf die `private`-Eigenschaften des bzw. der Objekte zugegriffen werden kann, weil die Klassenmethode im Bereich der Klasse selbst deklariert wurde. Und Klassenmethoden haben ja auch Zugriff auf die `private`-Eigenschaften einer Klasse.

In der Praxis können Sie die Deklaration im `public`-Bereich der Header-Datei *mensch.h* in der Klasse `Mensch` wie folgt vornehmen:

```
// mensch.h
#include <iostream>
#ifndef _MENSCH_H
#define _MENSCH_H
using namespace std;


class Mensch {
private:
   // Alles wie gehabt
   ...
public:
   // Alles wie gehabt
   ...
   int vergleiche_alter( const Mensch& p2 ) const;
};
#endif
```

Die Definition hingegen wird wieder in die Datei *mensch.cpp* ausgelagert und hat folgendes Aussehen:

```
// mensch.cpp
...
int Mensch::vergleiche_alter( const Mensch& p2 ) const {
   return alter - p2.alter;
}
```

Hieran lässt sich erkennen, wie Sie in der Methode direkt auf die `private`-Eigenschaften der Klasse zugreifen können. Aufgerufen wird diese Klassenmethode folgendermaßen:

```
Mensch person1( "Adam", 18, Mensch::MANN );
Mensch person2( "Eva", 19, Mensch::FRAU );
int ret;
...
ret = person1.vergleiche_alter( person2 );

if( ret > 0 ) {
   // ...
}
...
```

***this zum Zweiten ...**

Wenn Sie in einer Klassenmethode schreibend auf ein Objekt (als Argument) bzw. dessen Eigenschaften zugreifen wollen/müssen, muss dies natürlich ohne das Schlüsselwort `const` geschehen. Gewöhnlich verwendet man hierzu allerdings Zeiger statt Referenzen (für den schreibenden Zugriff), weil immer eine Adresse des Objekts angegeben werden muss.

Wollen Sie zum Beispiel die Eigenschaften eines Objekts kopieren oder zwei Objekte austauschen, können Sie mit dem `*this`-Zeiger das Objekt als »Ganzes« ansprechen oder die einzelnen Elemente (`this->eigenschaft`) separat. Fügen Sie zur Demonstration folgende zwei Klassenmethoden-Deklarationen in der Header-Datei *mensch.h* ein:

```
//mensch.h
...
class Mensch {
   ...
    // Mensch mit allen Eigenschaften kopieren
    void kopie_Mensch( Mensch *p );
    // Mensch mit allen Eigenschaften austauschen
    void tausche_Mensch( Mensch *p );
};
...
```

Die Definition in der Datei *mensch.cpp*:

```
void Mensch::kopie_Mensch( Mensch *p ) {
   Mensch tmp = *this;
   *p = tmp;
}
```

```
void Mensch::tausche_Mensch( Mensch *p ) {
   Mensch tmp = *p;
   *p = *this;
   *this = tmp;
}
```

Im Beispiel wurden die einzelnen Objekte mit `*this` als »Ganzes« verwendet, was Sie, wie bereits erwähnt, mit `this->` auch auf die einzelnen Eigenschaften anwenden könnten.

Beide Klassenmethoden, `kopie_Mensch()` und `tausche_Mensch()`, erwarten einen Zeiger, der hier immer neu beschrieben wird. Beide Methoden legen auch ein lokales Objekt `tmp` von der Klasse `Mensch` an. Das bedeutet, dass in der Funktion jeweils Konstruktor und Destruktor aufgerufen werden.

Bei der Methode `kopie_Mensch()` erhält das Objekt `tmp` die Adresse des Objekts durch `*this`. Damit haben Sie praktisch schon ein Objekt kopiert, das sich jetzt im lokalen Objekt `tmp` befindet. Damit nun auch die aufrufende Funktion mit dem aufgerufenen Zeiger als Parameter diese Kopie erhält, greifen Sie mit dem Zeiger `p` indirekt auf das Objekt `tmp` als »Ganzes« zurück, wodurch auch das Argument des Aufrufers auf ein gültiges Objekt (zunächst indirekt) verweist.

Ähnlich läuft es bei der Methode `tausche_Mensch()` ab, bei der die Eigenschaften zweier Objekte ausgetauscht werden. Auch hierbei wird ein lokales Objekt verwendet, das die Adresse des austauschbaren Objekts erhält. Anschließend übergeben Sie an `p` das Element des aktuellen Objekts (siehe letzten Abschnitt, »Klassenmethoden mit Objekten als Argumente«), was Sie mit `*this` gleich als »Ganzes« machen können. Am Ende erhält `*this` wieder das Objekt (genauer: die Adresse auf das Objekt), das Sie als Argument an die Funktion übergeben haben. Das muss in diesem Fall ein lokales Objekt sein.

In der Praxis lassen sich diese beiden Klassenmethoden wie folgt einsetzen:

```
// main.cpp
#include "mensch.h"

int main(void) {
   Mensch person1( "Adam", 18, Mensch::MANN );
   Mensch person2;
   Mensch person3( "Eva", 19, Mensch::FRAU );
   // Kopie erstellen
   person1.kopie_Mensch( &person2 );
   person1.print();
   person2.print();
   person3.print();
```

```
    // Eigenschaften der Objekte tauschen
    person1.tausche_Mensch( &person3 );
    person1.print();
    person2.print();
    person3.print();
    return 0;
}
```

Das Programm bei der Ausführung:

```
Adam ist von uns gegangen
Adam 18 Jahre (männlich)
Adam 18 Jahre (männlich)
Eva 19 Jahre (weiblich)

Eva ist von uns gegangen
Eva 19 Jahre (weiblich)
Adam 18 Jahre (männlich)
Adam 18 Jahre (männlich)
```

An diesem Beispiel können Sie außerdem erkennen, dass es manchmal hilfreich ist (auch für das Verständnis), den Destruktor etwas ausgeben zu lassen. In beiden Fällen wird dieser in den Klassenmethoden `kopie_Mensch()` und `tausche_Mensch()` ausgeführt – wo beide Male lokale Objekte verwendet wurden.

4.4.3 Objekte als Rückgabewert

Natürlich können Sie auch Objekte, wie bei anderen Typen, entweder als Zeiger, als Referenz oder als Objekt selbst zurückgeben lassen.

Allerdings gilt hierbei, wie auch schon bei der Parameterübergabe von Objekten an Funktionen, dass die Rückgabe als Kopie nicht unbedingt sehr sinnvoll erscheint – gerade bei größeren Objekten ist der Aufwand auf dem Stack nicht ganz unerheblich.

[»] **Hinweis**

Auch bei der Rückgabe von Objekten sollte man auf die Lebensdauer achten. Gerade bei der Rückgabe von Kopien bzw. Referenzen darf das Objekt nicht lokal beschränkt sein und sollte mindestens als `static` deklariert werden. Das kann man nicht oft genug erwähnen.

Es folgt ein einfaches Beispiel, wie Sie als Rückgabewert eine Kopie des Objekts zurückgeben lassen können. In diesem Beispiel wird eine globale Funktion `kopie_alter_Mensch()` erstellt, die die ältere von zwei Personen als Kopie an den Aufrufer als Rückgabewert zurückgibt. Schreiben Sie die Definition dieser

Funktion entweder in die Datei *main.cpp* oder in der Header-Datei *mensch.h* außerhalb der Klassendeklaration.

```
Mensch kopie_alter_Mensch( const Mensch& p1,
                           const Mensch& p2 ) {
   if( p1.vergleiche_alter(p2) > 0 ) {
      return p1;
   }
   else {
      return p2;
   }
}
```

Besser als diese Methode, bei der ein temporäres Objekt erzeugt und wieder zerstört werden muss, ist die Rückgabe von Zeigern oder Referenzen – im Hinblick auf die einfachere Verwendung sollte man immer, wenn möglich, Referenzen bevorzugen. Somit sieht diese Funktion mit der Rückgabe einer Referenz wie folgt aus:

```
const Mensch& kopie_alter_Mensch ( const Mensch& p1,
                                   const Mensch& p2 ) {
   if( p1.vergleiche_alter(p2) > 0 ) {
      return p1;
   }
   else {
      return p2;
   }
}
```

Die Verwendung dieser globalen Funktion kann wie folgt aussehen:

```
// main.cpp
#include "mensch.h"

int main(void) {
   Mensch person1( "Adam", 20, Mensch::MANN );
   Mensch person2( "Eva", 19, Mensch::FRAU );
   Mensch person3;
   person3 = kopie_alter_Mensch( person1, person2 );
   person3.print();
   return 0;
}
```

Das Ganze lässt sich auch noch komfortabler und objektorientierter ohne ein Objekt oder eine zusätzliche Referenz verwenden:

```
Mensch person1( "Adam", 20, Mensch::MANN );
Mensch person2( "Eva", 19, Mensch::FRAU );

cout << kopie_alter_Mensch(person1, person2).get_name()
     << " ist der Ältere von beiden!\n";
```

4.4.4 Klassen-Array (Array von Objekten)

Wollen Sie mehrere Objekte einer Klasse verwenden, so stehen Ihnen auch hier Arrays von Objekten zur Verfügung. Die Deklaration erfolgt wie bei der bereits gezeigten Syntax der Basistypen:

```
Klasse Bezeichner[AnzahlObjekte];
```

Deklarieren

Wollen Sie zum Beispiel ein Array `Mensch` mit zehn Objekten anlegen, können Sie dies folgendermaßen ausführen:

```
Mensch personen[10];
```

Wenn Sie keine explizite Initialisierung angeben, wird für jedes Objekt der Konstruktor (oder, falls keiner vorhanden ist, der Default-Konstruktor) aufgerufen.

Initialisieren

Auch die Initialisierung kann, wie schon bei den Basistypen bzw. den Strukturen, über eine Initialisierungsliste erfolgen. Wenn Sie wieder verschiedene Konstruktoren erstellt haben, dann kann jedes Element theoretisch mit einem anderen Konstruktor erzeugt werden:

```
// Ein Objekt der Klasse Mensch
Mensch ich("Jürgen", 30, Mensch::MANN );
// 10 Objekte der Klasse Mensch
Mensch personen[10] = {
   Mensch( "Adam", 20, Mensch::MANN ),
   Mensch( "Eva", 19, Mensch::FRAU ),
   Mensch( Mensch::FRAU ),
   Mensch( "Martin", 39 ),
   Mensch::FRAU,
   "Jenova",
   ich
};
```

Wenn Sie in Ihrer Klasse Konstruktoren mit nur einem Parameter angegeben haben, so können Sie in der Liste auch nur ein Argument angeben. Das ist natürlich nur sinnvoll, wenn die Klasse eindeutige Parameter besitzt. Sie können auch

ein bereits definiertes Objekt (wie hier `ich`) zur Liste hinzufügen. Die restlichen nicht definierten Objekte werden vom Konstruktor mit entsprechenden Werten vorbelegt.

Das Ganze lässt sich auch ohne die Angabe der Array-Länge verwenden. Nur werden dann so viele Elemente angelegt, wie in der Initialisierungsliste stehen:

```
// Ein Objekt der Klasse Mensch
Mensch ich("Jürgen", 30, Mensch::MANN );
// 7 Objekte der Klasse Mensch
Mensch personen[ ] = {
   Mensch( "Adam", 20, Mensch::MANN ),
   Mensch( "Eva", 19, Mensch::FRAU ),
   Mensch( Mensch::FRAU ),
   Mensch( "Martin", 39 ),
   Mensch::FRAU,
   "Jenova",
   ich
};
```

Ohne eine Angabe der Array-Länge werden hier sieben Objekte der Klasse `Mensch` angelegt.

Zugriff auf Klassenelemente

Der Zugriff auf die Klassenelemente (Klassenmethoden oder `public`-Eigenschaften) erfolgt ähnlich wie schon bei den normalen Objekten, nur muss hierbei der entsprechende Index in den eckigen Klammern mit angegeben werden.

```
Objekt[Index].Methode
```

Hierzu ein kleines Programmbeispiel, das die Verwendung von Klassen-Arrays in der Praxis demonstrieren soll:

```
// main.cpp
#include "mensch.h"

int main(void) {
   // Ein Objekt der Klasse Mensch
   Mensch ich("Jürgen", 30, Mensch::MANN );
   // 10 Objekte der Klasse Mensch
   Mensch personen[10] = {
      Mensch( "Adam", 20, Mensch::MANN ),
      Mensch( "Eva", 19, Mensch::FRAU ),
      Mensch( Mensch::FRAU ),
      Mensch( "Martin", 39 ),
      Mensch::FRAU,
```

```
      "Jenova",
      ich
   };
   // Anzahl der Objekte ermitteln
   cout << "Anzahl der Elemente: "
        << (sizeof(personen)/sizeof(Mensch)) << "\n";
   // Zugriff mit den Klassenmethoden
   personen[7].set_name("Georg");
   personen[7].set_alter( 50 );
   personen[7].set_geschlecht( Mensch::MANN );
   // Alles ausgeben
   for(int i=0; i < 10; i++ ) {
      personen[i].print();
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Anzahl der Elemente: 10
Adam 20 Jahre (männlich)
Eva 19 Jahre (weiblich)
Unbekannt 0 Jahre (weiblich)
Martin 39 Jahre (männlich)
Unbekannt 0 Jahre (weiblich)
Jenova 0 Jahre (weiblich)
Jürgen 30 Jahre (männlich)
Georg 50 Jahre (männlich)
Unbekannt 0 Jahre (männlich)
Unbekannt 0 Jahre (männlich)
```

4.4.5 Dynamische Objekte

Das Erzeugen dynamischer Objekte einer Klasse lässt sich, wie schon bei den bisher kennengelernten Typen, mit dem Operator `new` realisieren. Mit dem Reservieren von Speicher ist es aber bei den Klassen noch nicht getan. Hier wird zusätzlich noch der Konstruktor ausgeführt. Genauso sieht es beim Freigeben des Speicherplatzes aus. Neben der Freigabe des Speichers an den Heap wird auch noch der Destruktor für jedes reservierte Objekt aufgerufen.

Objekte dynamisch anlegen

Ansonsten ist die Syntax zur Reservierung von Speicher für Klassen dieselbe wie bei den Basistypen:

```
// Speicher für ein Objekt reservieren
new Klasse
```

```
// Speicher für ein Objekt reservieren und initialisieren
new Klasse( Initialisierungsliste )
// Speicher für n Objekte reservieren (dyn. Klassen-Array)
new Klasse[n]
```

Im ersten Beispiel wird Speicher für ein Objekt mit dem Standardkonstruktor aufgerufen (der Konstruktor ohne Parameter, falls vorhanden – ansonsten eine Minimalversion des Compilers). Im zweiten Fall wird ebenfalls Speicher für ein Objekt reserviert und ein entsprechender Konstruktor aufgerufen (abhängig von der Initialisierungsliste bzw. dem Vorhandensein eines solchen Konstruktors). Wird kein passender Konstruktor gefunden, wird eine Fehlermeldung ausgegeben. Beim dritten Beispiel reservieren Sie Speicher für `n` Objekte einer Klasse – also Speicher für ein dynamisches Klassen-Array. Auch hier wird für jedes einzelne Element der Standardkonstruktor aufgerufen.

Bezogen auf die Klasse `Mensch`, sieht das Beispiel so aus:

```
Mensch* MenschPtr;
// Speicher für ein Objekt (mit Initialisierungsliste)
MenschPtr = new Mensch("Adam", 20, Mensch::MANN );
```

Der Zeiger `MenschPtr` erhält nun bei erfolgreicher Speicherreservierung die Anfangsadresse eines Objekts der Klasse `Mensch`. Im Beispiel werden die einzelnen Eigenschaften gleich mit Werten initialisiert. Dies könnten Sie auch nachträglich (oder korrigierend) mit `MenschPtr` und dem Pfeiloperator machen. Wollen Sie zum Beispiel die Eigenschaft `alter` verändern, gehen Sie wie folgt vor:

```
// Alter verändern
MenschPtr->set_alter( 22 );
```

Mit dem Zeiger und dem Pfeiloperator können Sie auf alle `public`-Klassenmethoden und (falls vorhanden) `public`-Eigenschaften zugreifen.

Ähnlich können Sie auch Speicher für ein dynamisches Klassen-Array reservieren:

```
Mensch* MenschPtr;
// Speicher für Anzahl Objekte reservieren
MenschPtr = new Mensch[anzahl];
```

Die Objekte können Sie jetzt wie bei den gewöhnlichen Klassen-Arrays mit Werten initialisieren. Wichtig ist immer, dass Sie den (richtigen) Index verwenden. Dies können Sie entweder über die Klassenmethoden machen wie beispielsweise

```
MenschPtr[i].set_name( "Adam" );
MenschPtr[i].set_alter( 22 );
...
```

oder Sie verwenden einen Konstruktor (der Standardkonstruktor wurde ja bereits beim Anlegen mit `new` aufgerufen):

```
MenschPtr[i] = Mensch( "Adam", 22, Mensch::MANN );
i++;
MenschPtr[i] = Mensch( Mensch::FRAU );
...
```

Mehr dazu können Sie dem Abschnitt über Klassen-Arrays und Konstruktoren entnehmen – Abschnitt 4.4.4, »Klassen-Array (Array von Objekten)«.

Speicher freigeben

Analog sieht dies beim Freigeben des Speichers mit `delete` aus. Die Syntax dazu lautet:

```
// Speicher für ein Objekt wieder freigeben
delete KlassenPtr;
// Speicher für das Klassen-Array wieder freigeben
delete [] KlassenPtr;
```

Im ersten Beispiel wird der Speicher für ein reserviertes Objekt, auf das der Zeiger `KlassenPtr` verweist, an den Heap zurückgegeben. Damit auch alles zerstört wird, wird auch hier der Destruktor des Objekts aufgerufen. Im zweiten Fall geben Sie den Speicherplatz für ein dynamisch reserviertes Klassen-Array frei. Auch hierbei wird für jedes Element einzeln der Destruktor aufgerufen.

Bezogen auf Objekte der Klasse `Mensch`, ist das sehr einfach. Ein einzelnes Objekt geben Sie folgendermaßen frei:

```
delete MenschPtr;
```

Ebenso einfach wird der Speicher für ein ganzes Klassen-Array wieder abgebaut:

```
delete [] MenschPtr;
```

Hierzu nun ein Programmbeispiel, das das hier Beschriebene wieder in der Praxis demonstrieren soll. Wir verändern nur die Datei *main.cpp*, alles andere (*mensch.cpp*, *mensch.h*) bleibt unverändert:

```
// main.cpp
#include "mensch.h"

int main(void) {
   Mensch* MenschPtr;
   unsigned int anzahl;
   char name[30];
   unsigned int alter;
```

```
   bool geschlecht;

   MenschPtr = new Mensch("Adam", 20, Mensch::MANN );
   // Alter verändern
   MenschPtr->set_alter( 22 );
   MenschPtr->print();
   // Speicher wieder freigeben
   delete MenschPtr;

   cout << "Wie viele Personen sollen erzeugt werden: ";
   cin >> anzahl;
   // Speicher für Anzahl Objekte reservieren
   MenschPtr = new Mensch[anzahl];
   // Elemente initialisieren
   for(unsigned int i=0; i < anzahl; i++ ) {
      cout << "\nPerson " << i << " eingeben\n";
      cout << "Name                 : ";
      cin >> name;
      cout << "Alter                : ";
      cin >> alter;
      cout << "Geschlecht (m=0/w=1) : ";
      cin >> geschlecht;
      MenschPtr[i].set_name( name );
      MenschPtr[i].set_alter( alter );
      MenschPtr[i].set_geschlecht( geschlecht );
      /* Auch möglich wäre hierbei :
       * MenschPtr[i] = Mensch( name, alter, geschlecht );
       */
   }
   // Elemente ausgeben
   cout << "\nDie erzeugten Personen\n";
   for(unsigned int i=0; i < anzahl; i++ ) {
      MenschPtr[i].print();
   }
   // Speicher wieder freigeben
   delete [] MenschPtr;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Adam 22 Jahre (männlich)
Adam ist von uns gegangen
Wie viele Personen sollen erzeugt werden: 3
```

```
Person 0 eingeben
Name                 : Jürgen
Alter                : 30
Geschlecht (m=0/w=1) : 0

Person 1 eingeben
Name                 : Jonathan
Alter                : 3
Geschlecht (m=0/w=1) : 0

Person 2 eingeben
Name                 : Fatma
Alter                : 35
Geschlecht (m=0/w=1) : 1

Die erzeugten Personen
Jürgen 30 Jahre (männlich)
Jonathan 3 Jahre (männlich)
Fatma 35 Jahre (weiblich)
Fatma ist von uns gegangen
Jonathan ist von uns gegangen
Jürgen ist von uns gegangen
```

An der Ausgabe des Programms lässt sich gut erkennen, wie nach jedem `delete` der Destruktor für jedes Element aufgerufen wird.

Kein Speicherplatz mehr vorhanden

Wie bereits im Abschnitt zu `new` und `delete` (Abschnitt 2.1.5, »Dynamisch Speicherobjekte anlegen und zerstören – 'new' und 'delete'«) erwähnt, ist nicht immer garantiert, dass Sie einen angeforderten Speicher auch erhalten. Es kann passieren, dass der Heap irgendwann nicht mehr genügend zusammenhängenden Speicher reservieren kann. Wie man darauf reagieren kann, haben Sie auch bereits gelesen. Sofern Sie überhaupt nicht darauf reagieren, wird standardmäßig bei einem Fehlschlag der Speicheranforderung mit `new` das Programm beendet (abgebrochen).

Oft wird auch hier die »alte« Möglichkeit verwendet, indem man C-typisch den Rückgabewert auf `NULL` überprüft. Gibt eine Speicherreservierung `NULL` zurück, konnte kein Speicher mehr reserviert werden:

```
Mensch* MenschPtr;

for(int i = 0; ; i++ ) {
   MenschPtr = new (nothrow) Mensch[10000];
```

```
   if( MenschPtr == 0 ) {
      // Fehler beim Speicherreservieren
   }
}
```

Gewöhnlich reagiert man darauf, indem man

- eine Fehlermeldung auf den Standardfehler-Kanal (`cerr`) ausgibt
- gegebenenfalls die Daten sichert und das Programm beendet
- das Programm beendet
- nochmals versucht, Speicher vom Heap anzufordern

Fehler-Handle von »new«

Sofern Sie keine Vorkehrungen in Ihrem C++-Programm bezüglich des `new`-Operators treffen, wird ein Standard-new-Handler eingerichtet, der eine Exception (siehe Kapitel 6, »Exception-Handling«) auslöst. Wird diese Exception nicht abgefangen, wird das Programm beendet.

Hinweis **[«]**

Eine Exception ist eine unvorhergesehene Ausnahme. Die Ursachen einer solchen Exception können sehr vielfältig sein. Auf dieses Thema wird gezielt in Kapitel 6, »Exception-Handling«, eingegangen.

Mit dem »neuesten« ANSI-Standard bietet C++ einen recht komfortablen Handler an, mit dem man festlegen kann, was nach einem erfolglosen Aufruf von `new` passieren soll. Diesen Mechanismus können Sie mit der Funktion `set_new_handler`, die in der Header-Datei <*new*> definiert ist (und daher auch inkludiert werden muss), verwenden. Die Syntax lautet:

```
#include <new>
new_handler set_new_handler( new_handler _Pnew ) throw( );
```

Der `new`-Handler, den Sie dieser Funktion als Parameter übergeben, ist eine globale Funktion, die weder einen Parameter noch einen Rückgabewert besitzt. In dieser Funktion legen Sie fest, wie beim Fehlschlag einer Speicherreservierung fortgefahren werden soll. Mit der Funktion `set_new_handler` und der globalen Funktion als Parameter installieren Sie den `new`-Handler und ersetzen den Standard-new-Handler durch Ihren selbstgeschriebenen Handler. Hierzu das Programmbeispiel, das den `new`-Handler demonstriert:

```
// main.cpp
#include "mensch.h"
#include <cstdlib>
#include <new>
```

```
void heap_voll( void );

int main(void) {
   Mensch* MenschPtr;
   // Eigenen new-Handler installieren
   set_new_handler( heap_voll );
   for(int i = 0; ; i++ ) {
      MenschPtr = new (nothrow) Mensch[10000];
   }
   return 0;
}

// Eigener new-Handler
void heap_voll( void ) {
   cout << "Der Heap kann nicht mehr genügend"
        << " zusammenhängenden Speicher anbieten!\n"
        << "Das Programm wird beendet!\n";

   // Hier noch ggf. wichtige Arbeiten erledigen

   exit(1);
}
```

Wenn Sie das Beispiel testen wollen, sollten Sie sich ein wenig gedulden, bis der Heap »voll« ist (abhängig von der Größe des virtuellen Speichers und davon, wie das Betriebssystem damit umgeht). Sie können unter Windows in einem Task-Manager und unter Linux/*nix mit dem Befehl `top` beobachten, wie der Speicher Ihres Rechners immer voller wird (und der PC zwangsläufig langsamer reagiert). Wenn Sie keinen Speicher mehr bekommen, wird die Funktion `heap_voll()` ausgeführt und gibt die entsprechende Fehlermeldung aus.

4.4.6 Dynamische Klassenelemente

Natürlich ist die dynamische Speicherverwaltung nicht nur den Objekten vorbehalten, sondern auch den einzelnen Klassenelementen – man spricht auch von *dynamischen Elementen* oder *dynamischen Klasseneigenschaften*.

Zur Demonstration soll in der Klasse `Mensch` die Eigenschaft `name` dynamisch angelegt werden. Also statt

```
char name[30];
```

soll hier

```
char *name;
```

verwendet werden. Die Länge dieser Variablen soll erst zur Laufzeit vergeben werden. Dazu benötigen Sie natürlich eine weitere Variable, um die Länge der C-Zeichenkette zu speichern. Wir verwenden hierfür einen Integer `len`. Sie müssen also bei der Deklaration der Klasse in der Header-Datei *mensch.h* eine `private`-Eigenschaft verändern und eine hinzufügen:

```
// mensch.h
...
class Mensch {
private:
   // Eigenschaften der Klasse Mensch
   char *name;   // Speicher wird zur Laufzeit reserviert
   int len;      // Länge des Namens
   unsigned int alter;
   bool geschlecht; //0 = männlich; 1 = weiblich
public:
   // Alle public-Methoden und Eigenschaften wie gehabt
   ...
};
```

Es wurde bereits erwähnt, dass solche Umschreibungen von Programmen oft vorkommen. Um kompatibel zu älteren Programmversionen zu bleiben, sollten Sie, wenn möglich, die Schnittstellen der Funktionsköpfe nicht verändern. So können Sie sicherstellen, dass andere, die ein Update Ihrer Bibliothek erhalten, diese auch noch in den alten Hauptprogrammen verwenden können, ohne irgendwelche Veränderungen daran vornehmen zu müssen.

So werden auch keine Veränderungen der Schnittstellen notwendig. Sie müssen lediglich alle Funktionsdefinitionen, bei denen auf den C-String `name` bisher schreibend zugegriffen wurde, entsprechend der dynamischen Version anpassen.

Zunächst wären hier die Konstruktoren, die beim Anlegen eines Objekts immer den C-String mit einer Zeichenkette versehen. Betrachtet man zum Beispiel folgenden Konstruktor

```
// mensch.cpp
...
Mensch::Mensch( const char* n, unsigned int a, bool g ) {
   strncpy( name, n, sizeof(name)-1 );
   name[sizeof(name)] = '\0';
   alter = a;
   geschlecht = g;
}
```

dann muss der fett hervorgehobene Teil nur wie folgt umgeschrieben werden:

```
// mensch.cpp
...
Mensch::Mensch( const char* n, unsigned int a, bool g ) {
   // Länge des neuen C-Strings ermitteln
   len = strlen(n);
   // Speicher mit len+1 Bytes reservieren
   name = new char[len+1];
   // String in den neuen Speicher schreiben
   strcpy( name, n );
   alter = a;
   geschlecht = g;
}
```

Diese Änderungen müssen Sie an allen Konstruktoren in der Datei *mensch.cpp* bei deren Definition vornehmen. Hier die restlichen Konstruktoren:

```
// mensch.cpp
...
Mensch::Mensch( const char* n, unsigned int a )  {
   len = strlen(n);
   name = new char[len+1];
   strcpy( name, n );
   alter = a;
   // Zufallsgenerator wäre sinnvoller - aber ...
   if( a % 2 ) { // Gerade oder ungerade Zahl ...
      geschlecht = FRAU;
   }
   else {
      geschlecht = MANN;
   }
}

Mensch::Mensch( const char* n ) {
   len = strlen(n);
   name = new char[len+1];
   strcpy( name, n );
   // Neu geboren ... :-)
   alter = 0;
   // mehr Frauen braucht das Land ;-)
   geschlecht = FRAU;
}

Mensch::Mensch( ) {
   // Rabenvater oder Rabenmutter oder Tragödie ?
   len = strlen("Unbekannt");
   name = new char[len+1];
```

```
   strcpy( name, "Unbekannt" );
   // Neu geboren ... :-)
   alter = 0;
   // als Ausgleich wieder ein Mann ;-)
   geschlecht = MANN;
}

Mensch::Mensch( bool g ) {
   // Rabenvater oder Rabenmutter oder Tragödie ?
   len = strlen("Unbekannt");
   name = new char[len+1];
   strcpy( name, "Unbekannt" );
   // Neu geboren ... :-)
   alter = 0;

   // als Ausgleich wieder ein Mann ;-)
   geschlecht = g;
}
...
```

Als Nächstes folgen die Zugriffsmethoden, die auf den C-String `name` bisher schreibend zugegriffen haben; bei der Klasse `Mensch` sind dies nur zwei:

```
// mensch.cpp
...
void Mensch::erzeuge(const char* n, unsigned int a, bool g){
   // ... vom Konstruktor reservierten Speicher freigeben
   delete [] name;
   // Speicher für neuen Namen anfordern
   len = strlen(n);
   name = new char[len+1];
   strcpy( name, n );
   alter = a;
   geschlecht = g;
}
...
void Mensch::set_name( const char* n ) {
   // ... vom Konstruktor reservierten Speicher freigeben
   delete [] name;
   // Speicher für neuen Namen anfordern
   len = strlen(n);
   name = new char[len+1];
   strcpy( name, n );
}
```

Wird ein Objekt wieder freigegeben, müssen Sie auch den dynamisch reservierten Speicher für das Klassenelement explizit an den Heap zurückgeben. Ansonsten haben Sie ein Speicherleck (*Memory Leak*). Diese Aufgabe wird gewöhnlich in der Definition des Destruktors erledigt. In unserem Beispiel `Mensch` ist das dynamische Klassenelement ein C-String (also `char`-Array), den Sie mit `delete []` freigeben müssen. Der Destruktor in der Datei *mensch.cpp* sollte demnach so aussehen (bzw. muss erweitert werden):

```
Mensch::~Mensch( ) {
   cout << name << " ist von uns gegangen\n";
   // dynamisches Element wieder freigeben
   delete [] name;
}
```

Zum Testen können Sie nochmals die Datei *main.cpp* aus dem Abschnitt 4.4.5, »Dynamische Objekte«, verwenden, wo es darum ging, Objekte dynamisch anzulegen. An der Ausführung im Vordergrund und der Verwendung der Methoden des Programms hat sich nichts geändert. Nur die Interna der Klasse wurden verändert.

Hiermit wurde auch gleich ein weiterer Vorteil der OOP von C++ demonstriert: Der Wartungsaufwand durch die Verwendung von Klassen hält sich in Grenzen, da darauf geachtet wurde, dass die Schnittstellen (Kopf der Klassenmethoden) nicht verändert wurden.

4.4.7 Objekte kopieren (Kopierkonstruktor)

Sicherlich erinnern Sie sich noch an unsere Methode `kopiere_Mensch()`. Diese Methode hätten wir uns auch sparen können, weil der Compiler schon von Haus aus einen sogenannten Kopierkonstruktor zur Verfügung stellt. Damit werden alle Eigenschaften (Daten) einer Klasse in das neue Objekt kopiert:

```
Mensch person1( "Adam", 22, Mensch::MANN );
Mensch person2(person1);  // Gleich zu person2 = person1
```

Damit wird das Objekt `person2` durch einen Aufruf des Standard-Kopierkonstruktors mit den Eigenschaften von `person1` initialisiert. Natürlich können Sie einen solchen Kopierkonstruktor auch selbst deklarieren und definieren. Die Deklaration in der Header-Datei *mensch.h* als `public`-Element der Klasse `Mensch` würde folgendermaßen aussehen:

```
// mensch.h
...
public:
   ...
```

```
   // Kopierkonstruktor
   Mensch( const Mensch & );
   ...
```

Die Syntax der Deklaration eines solchen Kopierkonstruktors lautet also immer:

```
Klasse ( const Klasse & );
```

Eine Definition in *mensch.cpp* ist nicht unbedingt nötig und kann auch in *mensch.h* mit zwei leeren geschweiften Klammern erfolgen. Wenn Sie den Effekt des Kopierkonstruktors testen wollen, können Sie auch folgende Definition dazu verwenden:

```
Mensch::Mensch( const Mensch & ) {
   cout << "Kopier-Konstruktor aufgerufen\n";
}
```

Allerdings können Sie diesen Kopierkonstruktor nur bei den Beispielen verwenden, die vor Abschnitt 4.4.6, »Dynamische Klassenelemente«, beschrieben wurden. Durch die Verwendung von dynamischen Klassenelementen sind Probleme vorprogrammiert, weil Zeiger von verschiedenen Objekten auf denselben Speicherbereich zeigen würden. Im Beispiel der Klasse `Mensch` würde beim Kopieren zweier Objekte auch dieselbe Adresse des Namens »kopiert«. Sobald aber eines der Objekte zerstört bzw. verändert (vergrößert, verkleinert) wird, treten Probleme auf. Daher kommen Sie bei Objekten mit dynamischen Klassenelementen nicht drum herum, einen eigenen Kopierkonstruktor zu definieren, der nicht den Zeiger, sondern die Eigenschaften (Daten) mitkopiert. Bezogen auf unsere Klasse `Mensch`, sieht die Verwendung (mit Deklaration und Definition in den entsprechenden Dateien) des Kopierkonstruktors wie folgt aus:

```
// mensch.h
...
// Deklaration des Kopierkonstruktors
Mensch( const Mensch &person );
...

// mensch.cpp
...
// Definition des Kopierkonstruktors
Mensch::Mensch( const Mensch& person ) {
   len = person.len;
   name = new char[len+1];
   strcpy( name, person.name );
   alter = person.alter;
   geschlecht = person.geschlecht;
}
...
```

```
// main.cpp
...
Mensch person1( "Adam", 22, Mensch::MANN );
// Verwendung des Kopierkonstruktors
Mensch person2(person1);
```

4.4.8 Dynamisch erzeugte Objekte kopieren (»operator=()«)

Im letzten Abschnitt wurde kurz erwähnt, dass eine Zuweisung wie

```
Objekt2 = Objekt1;
```

etwa dasselbe ist wie

```
Klasse Objekt1( Initialisierungs-Liste );
Klasse Objekt2(Objekt1);
```

Bei dieser Methode (es ist tatsächlich eine echte Methode) handelt es sich um eine Standardzuweisung. Das Prinzip entspricht genau dem des Standard-Kopierkonstruktors. Dies bedeutet allerdings auch, dass hier die bekannten Probleme auftreten, wenn ein Objekt dynamische Klassenelemente verwendet. Daher muss auch die Standardzuweisung bei der Verwendung von dynamischen Elementen durch eine selbstdefinierte Zuweisung ersetzt werden.

Dabei muss auf das Prinzip der Operator-Überladung zurückgegriffen werden, das erst in Abschnitt 4.5, »Operatoren überladen«, ausführlicher behandelt wird. Trotzdem soll kurz darauf eingegangen werden, wie Sie den Zuweisungsoperator = der Klasse `Mensch` überladen können.

Zuweisungen werden immer von der Methode `operator=()` ausgeführt. Das bedeutet, dass eine Standardzuweisung wie

```
Objekt2 = Objekt1;
```

dasselbe bedeutet wie

```
Objekt2.operator=(Objekt1);
```

Hierzu können Sie jetzt in der Praxis die Methode `operator=()` der Klasse `Mensch` wie folgt überladen:

```
// mensch.h
...
public:
   ...
   // Zuweisungsoperator überladen - Deklaration
   Mensch& operator=( const Mensch& person );
   ...
```

```
// mensch.cpp
...
// Zuweisungsoperator überladen - Definition
Mensch& Mensch::operator=( const Mensch& person ) {
   // Auf Selbst-Zuweisung überprüfen
   if( this == &person ) {
      return *this;
   }
   else {
      len = person.len;
      name = new char[len+1];
      strcpy( name, person.name );
      alter = person.alter;

      geschlecht = person.geschlecht;
      return *this;
   }
}
...

// main.cpp
...
Mensch person1( "Adam", 22, Mensch::MANN );
// Zuweisungsoperator überladen - Verwendung
Mensch person2 = person1;
...
```

Das nur in Kürze, falls Sie den Zuweisungsoperator in Abschnitt 4.4.7, »Objekte kopieren (Kopierkonstruktor)«, verwendet und sich gewundert haben, warum dies nicht funktioniert. Die Operator-Überladung wird in Abschnitt 4.5, »Operatoren überladen«, erläutert.

4.4.9 Standardmethoden (Überblick)

Auf den letzten Seiten dürfte Ihnen aufgefallen sein, dass Ihnen der Compiler mehrere Standardmethoden zur Verfügung stellt, die jederzeit explizit durch eigene Methoden ersetzt werden konnten. Diese vier Methoden sind:

- Standardkonstruktor
- Destruktor
- Kopierkonstruktor
- Standardzuweisung durch den Zuweisungsoperator

4.4.10 Objekte als Elemente (bzw. Eigenschaften) in anderen Klassen

Es lassen sich neben den gewöhnlichen Datentypen natürlich auch Objekte anderer Klassen selbst als Eigenschaften (Daten) einer Klasse verwenden. Als Beispiel dient eine einfache Klasse `Haustier`, die auf das Grundlegende beschränkt ist und anschließend in der Klasse `Mensch` verwendet werden soll. Hier die Header-Datei *htier.h*, bei der die implizite `inline`-Definition mit »eingepackt« wurde, so dass alles zur Verfügung steht:

```
// htier.h
#include <iostream>
#include <cstring>
#ifndef _HTIER_H
#define _HTIER_H
using namespace std;

class Haustier {
private:
   // Eigenschaften der Klasse Haustier
   char sorte[30];
   char rasse[30];
   char name[25];

public:
   // Konstruktoren
   // Default-Konstruktor
   Haustier( ) {
      strncpy( sorte, "?", sizeof("?") );
      strncpy( rasse, "?", sizeof("?") );
      strncpy( name, "?", sizeof("?") );
   }
   Haustier( const char* s, const char* r, const char* n ) {
      strncpy( sorte, s, sizeof(sorte)-1 );
      strncpy( rasse, r, sizeof(rasse)-1 );
      strncpy( name, n, sizeof(name)-1 );
   }
   // Destruktor
   ~Haustier( ) { }

   // Fähigkeiten (Methoden) der Klasse Haustier
   // Hier nur auf Zugriffsmethoden beschränkt
   const char* get_sorte() const { return sorte; }
   const char* get_rasse() const { return rasse; }
   const char* get_name() const { return name; }
   void set_sorte( const char* s ) {
```

```
      strncpy( sorte, s, sizeof(sorte)-1 );
   }
   void set_rasse( const char* r ) {
      strncpy( rasse, r, sizeof(rasse)-1 );
   }
   void set_name( const char* n ) {
      strncpy( name, n, sizeof(name)-1 );
   }
};
#endif
```

In der Praxis könnten Sie diese Klasse `Haustier` wie die Klasse `Mensch` im Hauptprogramm einsetzen:

```
// main_htier.cpp
#include "htier.h"
int main(void) {
   Haustier tier1;
   Haustier tier2( "Hund", "Jack-Russell", "Blinki" );
   tier1.set_sorte( "Katze" );
   tier1.set_rasse( "Longhair" );
   tier1.set_name( "Maunzi" );

   cout << tier1.get_name() << ' '
        << tier1.get_sorte() << ' '
        << tier1.get_rasse() << '\n';

   cout << tier2.get_name() << ' '
        << tier2.get_sorte() << ' '
        << tier2.get_rasse() << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
Maunzi Katze Longhair
Blinki Hund Jack-Russell
```

Allerdings entspricht dieses Beispiel nicht unseren Absichten in diesem Abschnitt. Wir wollen jetzt die Klasse `Haustier` in einer anderen Klasse verwenden. Diese Klasse arbeitet dann mit dem Objekt `Haustier` – man spricht von einer *Hat-Beziehung*. Natürlich verwenden wir wieder die Klasse `Mensch`, da ein Mensch auch gerne ein Haustier hält.

Hierzu müssen Sie zunächst bei den Eigenschaften (Daten) der Klasse ein neues Objekt vom Typ `Haustier` einfügen:

```
// mensch.h
#include <iostream>
#include "htier.h"
#ifndef _MENSCH_H
#define _MENSCH_H
using namespace std;

class Mensch {
private:
   // Eigenschaften der Klasse Mensch
   char *name;
   int len;
   unsigned int alter;
   bool geschlecht;
   Haustier tier;
public:
   ...
```

Hiermit haben Sie zunächst ein privates Objekt `tier` der Klasse `Haustier` zu den Klasseneigenschaften von `Mensch` hinzugefügt. Wenn Sie ein neues Objekt `Mensch` anlegen, wird der Konstruktor (bzw. Default-Konstruktor, abhängig von der Initialisierungsliste) aufgerufen. Befindet sich in einem solchen Objekt ein weiteres Teilobjekt (wie im Beispiel die Klasse `Haustier`), so wird auch der Default-Konstruktor des Teilobjekts aufgerufen. Wenn Sie also ein Objekt der Klasse `Mensch` wie folgt definieren

```
Mensch person;
```

wird neben dem Default-Konstruktor für das Objekt `Mensch` auch der Default-Konstruktor für `Haustier` aufgerufen. In unserem Beispiel sieht der Default-Konstruktor für `Haustier` so aus:

```
   Haustier( ) {
      strncpy( sorte, "?", sizeof("?") );
      strncpy( rasse, "?", sizeof("?") );
      strncpy( name, "?", sizeof("?") );
   }
```

Alle Werte werden mit einem Fragezeichen belegt. Das können Sie natürlich auch anders machen. Neben dem Default-Konstruktor wurde in der Klasse `Haustier` noch ein weiterer definiert, dem Sie die komplette Initialisierungsliste der Klasseneigenschaften übergeben können.

Zurück zur Klasse `Mensch` – wenn Sie ein Objekt der Klasse `Mensch` anlegen, werden Sie dies auch mit einer Initialisierungsliste machen wollen. Somit benötigen Sie für die Klasse `Mensch` einen Konstruktor, der das Teilobjekt der Klasse `Haus-`

tier beachtet und es ebenfalls mit entsprechenden Werten versieht, also nicht mit den Default-Konstruktor aufruft.

Hierzu zwei Beispiele von expliziten Konstruktoren, die ein Teilobjekt auch explizit initialisieren. Dies lässt sich einfacher realisieren, als es den Anschein hat:

```
// mensch.h
...
public:
// Konstruktoren
Mensch( const char* n, unsigned int a, bool g,
        const char* s, const char* r, const char* na );
Mensch( const char* n, unsigned int a, bool g,
        const Haustier t);
...
```

Sie haben nun die Möglichkeit, beim Anlegen eines Objekts das Teilobjekt tier der Klasse Haustier indirekt oder direkt mit Werten zu initialisieren. Hierzu die Definition der beiden Konstruktoren in der Datei *mensch.cpp*:

```
// mensch.cpp
...
Mensch::Mensch( const char* n, unsigned int a, bool g,
           const char* s, const char* r, const char* na ) {
   len = strlen(n);
   name = new char[len+1];
   strcpy( name, n );
   alter = a;
   geschlecht = g;
   tier = Haustier ( s, r, na );
}

Mensch::Mensch( const char* n, unsigned int a, bool g,
           const Haustier t ) {
   len = strlen(n);
   name = new char[len+1];
   strcpy( name, n );
   alter = a;
   geschlecht = g;
   tier = t;
}
...
```

Hinweis [«]

Es sollte noch erwähnt werden, dass immer erst das Teilobjekt vor dem eigentlichen Objekt aufgebaut wird, damit der Compiler weiß, welche Argumente in der Initialisierungsliste zu welchem Teilobjekt gehören.

Gegebenenfalls werden Sie einige Methoden noch etwas ergänzen müssen, damit auch das Teilobjekt `tier` korrekt übernommen wird. Das betrifft zum Beispiel den Kopierkonstruktor, da sonst beim Kopieren nicht die Haustiere übernommen werden, sondern lediglich neue, die mit Fragezeichen erzeugt würden:

```
Mensch::Mensch( const Mensch& person ) {
   len = person.len;
   name = new char[len+1];
   strcpy( name, person.name );
   alter = person.alter;
   geschlecht = person.geschlecht;
   tier = person.tier;
}
```

Jetzt fehlt nur noch der Zugriff auf die `public`-Elemente der Klasse `Haustier` aus der Klasse `Mensch` heraus. Dieser erfolgt aus den Klassenmethoden der Klasse `Mensch`, indem zuerst der Name des Teilobjekts (hier »tier«) angegeben wird, dann folgt der Punktoperator und zuletzt das entsprechende (`public`-)Klassenelement. Im Prinzip erfolgt der Zugriff genauso, wie Sie bisher auf die `public`-Elemente einer Klasse von außen zugegriffen haben. Als Beispiel habe ich der Klasse `Mensch` eine Methode `printall()` hinzugefügt, die alle Daten mitsamt den `Haustier`-Daten ausgibt. Die herkömmliche Funktion `print()` bleibt nach wie vor der Ausgabe der Klasse `Mensch` vorbehalten:

```
// mensch.h
...
public:
...
// Ausgabe aller Eigenschaften mit der Klasse Haustier
void printall( void );

// mensch.cpp
...
void Mensch::printall ( void ) {
   cout << name << " " << alter << " Jahre (";
   test_geschlecht();
   cout << " -> Haustier: "
        << tier.get_sorte() << " (Rasse:"
        << tier.get_rasse() << "/Name:"
        << tier.get_name()  << "))\n";
}
...
```

Hinweis [«]

Auch hier sei angemerkt, dass durch die Änderung der Klasse `Mensch` mit der Erweiterung eines Objekts der Klasse `Haustier` keine Probleme mit den Hauptprogrammen entstehen, da auf die Beibehaltung der Schnittstelle geachtet wurde. Sie können also durchaus ein Beispiel der »älteren« Sorte testen, und es sollte ohne Probleme funktionieren.

Hierzu nun unser Hauptprogramm, das die Verwendung von Teilobjekten in anderen Klassen (hier des Teilobjekts `tier` der Klasse `Haustier` in der Klasse `Mensch`) demonstriert:

```
// main.cpp
#include "mensch.h"

int main(void) {
   Mensch person1( "Adam", 22, Mensch::MANN,
                   "Hund", "Dackel", "Waldi" );
   Haustier cat( "Katze", "Perser", "Minzi" );
   Mensch person2( "Eva", 19, Mensch::FRAU, cat );
   Mensch person3;

   person1.printall();
   person2.printall();
   person3.printall();
   return 0;
}
```

Das Programm bei der Ausführung:

```
Adam 22 Jahre (männlich -> Haustier: Hund (Rasse:Dackel/Name:Waldi))
Eva 19 Jahre (weiblich -> Haustier: Katze (Rasse:Perser/Name:Minzi))
Unbekannt 0 Jahre (männlich -> Haustier: ? (Rasse:?/Name:?))
```

Jetzt stellt sich die Frage, wie man von außen (also z. B. außerhalb der `main`-Funktion) auf die Methoden bzw. Klassenelemente des Teilobjekts zugreifen kann.

Um von außen (außerhalb der Klasse) auf die Elemente eines Teilobjekts zuzugreifen, muss zuerst der Name des Objekts verwendet werden, gefolgt von einem Punkt und dem Namen des Teilobjekts sowie einem weiteren Punkt mit dem abschließenden gewünschten Element des Teilobjekts. Dies funktioniert allerdings nur dann, wenn das Teilobjekt `public` ist und nicht `private`, wie im Beispiel zuvor. Sofern das Teilobjekt in der Klasse im `private`-Bereich untergebracht ist, haben Sie von außen keinen Zugriff auf die Elemente des Teilobjekts.

Um also Folgendes realisieren zu können, müssen Sie das Teilobjekt `tier` (in der Header-Datei *mensch.h*) in der Klasse `Mensch` aus dem `private`-Bereich entfernen und im `public`-Bereich einfügen:

```
// main.cpp
#include "mensch.h"

int main(void) {
   Mensch person1( "Adam", 22, Mensch::MANN,
                   "Hund", "Dackel", "Waldi" );
   Haustier cat( "Katze", "Perser", "Minzi" );
   Mensch person2("Eva", 19, Mensch::FRAU, cat );

   cout << "Das Haustier von " << person1.get_name()
     << " heißt " << person1.tier.get_name()
     << " und ist ein " << person1.tier.get_sorte() << '\n';

   person2.tier.set_name( "Minka" );

   cout << person2.tier.get_name() << '\n';
   return 0;
}
```

Auf der anderen Seite können Sie durch die private Definition eines Teilobjekts die Schnittstelle des Teilobjekts nach außen »verstecken«, so dass Sie bei einem Zugriff auf Elemente eines Teilobjekts gezwungen sind, in der Klasse hierfür eine eigene Schnittstelle zur Verfügung zu stellen, was in der Regel zu empfehlen ist, da hierbei nicht das OOP-Schema unterlaufen werden muss.

4.4.11 Teilobjekte initialisieren

Bisher wurde bei der Erzeugung von Objekten, die Teilobjekte enthielten, zunächst der Default-Konstruktor (der Teilobjekte) aufgerufen und das Teilobjekt mit den Standardwerten belegt. Erst anschließend erhielt das Teilobjekt bei der Zuweisung die richtigen Werte. Ein Vorgang mit doppelter Arbeit, der nicht unbedingt sein muss. C++ bietet mit einem speziellen Elementinitialisierer eine Möglichkeit an, Teilobjekte einer Klasse explizit mit Werten zu initialisieren.

Das soll am Beispiel des Konstruktors `Mensch`, den Sie im letzten Abschnitt verwendet haben, gezeigt werden:

```
// mensch.cpp
...
Mensch::Mensch( const char* n, unsigned int a, bool g,
           const char* s, const char* r, const char* na ) {
```

```
   len = strlen(n);
   name = new char[len+1];
   strcpy( name, n );
   alter = a;
   geschlecht = g;
   tier = Haustier ( s, r, na );
}
```

Die Definition dieses Konstruktors können Sie nun wie folgt mit einer zusätzlichen Initialisierungsliste belegen:

```
// mensch.cpp
...
Mensch::Mensch( const char* n, unsigned int a, bool g,
           const char* s, const char* r, const char* na )
           :tier( s, r, na )      {
   len = strlen(n);
   name = new char[len+1];
   strcpy( name, n );
   alter = a;
   geschlecht = g;
}
```

Hinweis [«]

Bitte beachten Sie, dass bei der Verwendung von Elementinitialisierern wirklich nur die Definition verändert werden muss. Die Deklaration des Konstruktors bleibt gleich!

Durch den oben verwendeten Elementinitialisierer wird jetzt gleich der passende Konstruktor für die Eigenschaften aufgerufen und nicht mehr der Default-Konstruktor. Die Liste enthält die Namen der Eigenschaften (Datenelemente) mit den entsprechenden Anfangswerten. Die Elementinitialisierer werden hinter dem Funktionskopf getrennt mit einem Doppelpunkt angegeben. Die einzelnen Elemente werden durch ein Komma getrennt.

Dies funktioniert natürlich auch bei dem Konstruktor, bei dem Sie im Beispiel des Abschnitts zuvor als viertes Argument das komplette Teilobjekt `Haustier` übergeben haben:

```
// mensch.cpp
...
Mensch::Mensch( const char* n, unsigned int a, bool g,
                const Haustier t ):tier(t) {
   len = strlen(n);
   name = new char[len+1];
   strcpy( name, n );
```

```
   alter = a;
   geschlecht = g;
}
```

Auch wenn Ihnen diese Möglichkeit, Teilobjekte zu initialisieren, ein wenig überflüssig erscheinen mag, so bietet sie doch große Vorteile:

- Das Laufzeitverhalten des Programms verbessert sich. Es muss nicht unnötig Speicherplatz angefordert (Default-Konstruktor) und wieder freigegeben werden, um anschließend erneut Speicher anzufordern (Konstruktor), um das Teilobjekt mit Werten zu initialisieren.
- Als Datenelemente können auch konstante Objekte und Referenzen verwendet werden.

4.4.12 Klassen in Klassen verschachteln

Zwar wird in der Praxis kaum davon Gebrauch gemacht, aber es sollte dennoch erwähnt werden, dass es auch möglich ist, Klassen zu verschachteln. Dies hieße dann, dass eine Klasse lokal innerhalb einer Klasse deklariert wird. Somit gilt die Klasse nur in ihrem Gültigkeitsbereich:

```
// classA.cpp
...
class A {
   public:
   ...
   private:
   class B {  // lokale Klasse
      // ...
   }
};
```

Ein mögliches Anwendungsgebiet verschachtelter Klassen wäre die Verwendung von Klassen, die mit gleichem Namen eine globale Gültigkeit haben. So könnten Sie in einer Klasse eine interne Klasse vereinbaren, ohne dass es zu Namenskonflikten kommt, weil die lokale Gültigkeit Vorrang hat. Im Grunde sollten Klassen, die innerhalb von Klassen verwendet werden, auch nur dort gelten, daher werden diese ja auch lokal vereinbart.

Im Codeausschnitt *classA.cpp* kann, so definiert, keine der Klassen auf die Elemente der anderen Klasse zugreifen. Also kann weder `A` auf `B` noch `B` auf Elemente von `A` zugreifen. Wollen Sie, dass `A` Zugriff auf `B` hat, müssen Sie nur die Klasse `B` im `public`-Teil von `A` vereinbaren und `A` nur zu einem `friend` (siehe Abschnitt 4.4.16, »Friend-Funktionen bzw. friend-Klassen«) von `B` erklären:

```
// classA.cpp
...
class A {
   public:
   class B {  // lokale Klasse
      friend class A;
   }
   private:
   ...
};
```

Da sich der Umfang und die Unübersichtlichkeit verschachtelter Klassen schnell ins Chaotische entwickeln können, empfiehlt es sich in der Praxis, die verschachtelte Klasse in der umgebenden Klasse zu deklarieren, aber außerhalb zu definieren. Bei der Definition der verschachtelten Klasse muss die innere Klasse über den Namen der äußeren Klasse und den Scope-Operator verwendet werden:

```
// classA.cpp
...
class A {
   public:
   // Deklaration der Klasse B
   class B;
   private:
   ...
};

...
// Definition der Klasse B
class A::B {
   // Definition
}
```

4.4.13 Konstante Klasseneigenschaften (Datenelemente)

Wollen Sie in einer Klasse Eigenschaften verwenden, die nicht mehr verändert werden können, so wird diese mit `const` deklariert. Beispielsweise deklarieren Sie in der Klasse `Mensch` das Teilobjekt `tier` der Klasse `Haustier` als konstantes Objekt:

```
// mensch.h
...
class Mensch {
private:
   // Eigenschaften der Klasse Mensch
   char *name;
   int len;
```

```
   unsigned int alter;
   bool geschlecht; //0 = männlich; 1 = weiblich
   const Haustier tier;

public:
   ...
}
```

Sie müssen allerdings darauf achten, dass gleich bei der Initialisierung des Objekts der richtige Konstruktor aufgerufen wird, weil bei konstanten Daten keine späteren Zuweisungen mehr möglich sind.

Das bedeutet, dass gleich bei der Definition des Konstruktors der Klasse jedes `const`-Element einen Elementinitialisierer benötigt. Folgender Konstruktor würde zum Beispiel zu einer Fehlermeldung des Compilers führen:

```
Mensch::Mensch( const char* n, unsigned int a, bool g,
                const Haustier t ) {
   len = strlen(n);
   name = new char[len+1];
   strcpy( name, n );
   alter = a;
   geschlecht = g;
   // !!! Fehler !!! const-Elemente bereits
   // vom Default-Konstruktor vorbelegt
   tier = t;
}
```

Wie Sie bereits erfahren haben, wird beim Vereinbaren eines Objekts auch der Konstruktor eines gegebenenfalls vorhandenen Teilobjekts aufgerufen. In unserem Beispiel würde praktisch beim Erzeugen eines neuen Objekts `Mensch` schon der Default-Konstruktor aufgerufen, der alle Werte mit Fragezeichen belegt. Daher führt die anschließende Zuweisung zu einem Fehler, weil das `const`-Element bereits mit Werten belegt ist. Die Fehlermeldung verschwindet, wenn Sie hierbei die Zeile

```
tier = t;
```

entfernen, wodurch das Teilobjekt `tier` mit den Werten (Fragezeichen) des Default-Konstruktors vorbelegt wird. Dies ist allerdings nicht im Sinne des Erfinders. Daher benötigt auch der Konstruktor einen Elementinitialisierer, um den Aufruf des Default-Konstruktors gegebenenfalls zu umgehen. In unserem Fall würde dies Folgendes bedeuten:

```
Mensch::Mensch( const char* n, unsigned int a, bool g,
                const Haustier t ):tier(t) {
```

```
   len = strlen(n);
   name = new char[len+1];
   strcpy( name, n );
   alter = a;
   geschlecht = g;
}
```

Dasselbe gilt übrigens auch für die elementaren Datentypen, die Sie ebenfalls in einer Klasse als konstant definieren können. Auch hier lässt sich der Elementinitialisierer einsetzen – und das für alle Typen:

```
class A {
   private:
   const int iwert;
   ...
   public:
   // Konstruktor
   A( int k_iwert, ... ):iwert(k_iwert) {
      // ...
   }
}
```

4.4.14 Statische Klasseneigenschaften (Datenelemente)

Wollen Sie bei Objekten einer Klasse nicht nur die Eigenschaften für ein Objekt allein verwenden, sondern soll eine Eigenschaft gemeinsam (mit anderen Objekten) genutzt bzw. geteilt werden, müssen Sie diese Klasseneigenschaften als `static` vereinbaren. Diese mit `static` deklarierte Variable ist für alle anderen Objekte nur einmal im Speicher vorhanden.

Vorwiegend werden solche statischen Eigenschaften verwendet, um bestimmte Informationen einer Klasse anzuzeigen, beispielsweise die Anzahl der bereits erzeugten Objekte einer Klasse oder zwischengespeicherte Höchst- bzw. Niedrigwerte. Außerdem können statische Eigenschaften verwendet werden, um Daten temporär zwischenzuspeichern, so dass diese Daten wiederum den anderen Objekten zur Verfügung stehen.

Um ein statisches Element zu deklarieren, ist so vorzugehen, wie es schon in Bezug auf das Schlüsselwort `static` beschrieben wurde:

```
static typ bezeichner;
```

Bei unserer Klasse `Mensch` würde sich das Beispiel ganz gut eignen, um die Weltbevölkerung zu zählen:

```
// mensch.h
...
class Mensch {
   private:
      // Eigenschaften der Klasse Mensch
      char *name;   // Speicher wird zur Laufzeit reserviert
      int len;      // Länge des Namens
...
...
   public:
      static int anzahlMensch;
      // Deklaration der Konstruktoren
      ...
```

Dieses so statisch vereinbarte Klassenelement einer Klasse belegt sofort einen Speicherplatz, auch wenn noch kein Objekt dieser Klasse existiert. Deshalb müssen Sie diese Variable wie die Klassenmethoden auch außerhalb der Klasse in einer Datei definieren und gegebenenfalls initialisieren. Hierzu werden auch die Klasse und der Scope-Operator `::` benötigt. In unserem Beispiel sieht die Definition in der Datei *mensch.cpp* so aus:

```
// mensch.cpp
#include <iostream>
#include <cstring>
#include "mensch.h"
using namespace std;

// Definition und Initialisierung
int Mensch::anzahlMensch = 0;
...
```

Beachten Sie außerdem, dass hierbei nicht mehr das Schlüsselwort `static` mit angegeben wird. Natürlich benötigen Sie jetzt auch Methoden, die auf den Wert dieser statischen Variablen zugreifen. In unserem Fall verwenden wir die Konstruktoren, bei denen der Wert um eins inkrementiert wird, und den Destruktor, der den Wert von `anzahlMensch` um eins dekrementiert. Außerdem wird in diesem Beispiel bei der Verwendung des Kopierkonstruktors dem überladenden Zuweisungsoperator und der Klassenmethode `erzeuge` der Wert von `anzahlMensch` inkrementiert.

```
Mensch::Mensch( const char* n, unsigned int a, bool g,
       const char* s, const char* r, const char* na ):
       tier( s, r, na )      {
   len = strlen(n);
   name = new char[len+1];
```

```
   strcpy( name, n );
   alter = a;
   geschlecht = g;
   ++anzahlMensch;
}

Mensch::Mensch( const char* n, unsigned int a, bool g,
                const Haustier t ):tier(t) {
   len = strlen(n);
   name = new char[len+1];
   strcpy( name, n );
   alter = a;
   geschlecht = g;
   ++anzahlMensch;
}

Mensch::Mensch( const char* n, unsigned int a, bool g ) {
   len = strlen(n);
   name = new char[len+1];
   strcpy( name, n );
   alter = a;
   geschlecht = g;
   ++anzahlMensch;
}

Mensch::Mensch( const char* n, unsigned int a )  {
   len = strlen(n);
   name = new char[len+1];
   strcpy( name, n );
   alter = a;
   // Zufallsgenerator wäre sinnvoller - aber ...
   if( a % 2 ) { // Gerade oder ungerade Zahl ...
      geschlecht = FRAU;
   }
   else {
      geschlecht = MANN;
   }
   ++anzahlMensch;
}

Mensch::Mensch( const char* n ) {
   len = strlen(n);
   name = new char[len+1];
   strcpy( name, n );
   // Neu geboren ... :-)
```

```
   alter = 0;
   // mehr Frauen braucht das Land ;-)
   geschlecht = FRAU;
   ++anzahlMensch;
}

Mensch::Mensch( ) {
   // Rabenvater oder Rabenmutter oder Tragödie ?
   len = strlen("Unbekannt");
   name = new char[len+1];
   strcpy( name, "Unbekannt" );
   // Neu geboren ... :-)
   alter = 0;
   // als Ausgleich wieder ein Mann ;-)
   geschlecht = MANN;
   ++anzahlMensch;
}

Mensch::Mensch( bool g ) {
   // Rabenvater oder Rabenmutter oder Tragödie ?
   len = strlen("Unbekannt");
   name = new char[len+1];
   strcpy( name, "Unbekannt" );
   // Neu geboren ... :-)
   alter = 0;
   // als Ausgleich wieder ein Mann ;-)
   geschlecht = g;
   ++anzahlMensch;
}

Mensch::Mensch( const Mensch& person ) {
   len = person.len;
   name = new char[len+1];
   strcpy( name, person.name );
   alter = person.alter;
   geschlecht = person.geschlecht;
   tier = person.tier;
   ++anzahlMensch;
}

// Definition des Destruktors
Mensch::~Mensch( ) {
   cout << name << " ist von uns gegangen\n";
   delete [] name;
   --anzahlMensch;
}
```

```
// Zuweisungsoperator überladen
Mensch& Mensch::operator=( const Mensch& person ) {
   // Auf Selbst-Zuweisung überprüfen
   if( this == &person ) {
      return *this;
   }
   else {
      len = person.len;
      name = new char[len+1];
      strcpy( name, person.name );
      alter = person.alter;
      geschlecht = person.geschlecht;
      ++anzahlMensch;
      return *this;
   }
}

void Mensch::erzeuge(const char* n,unsigned int a,bool g ) {
   len = strlen(n);
   name = new char[len+1];
   strcpy( name, n );
   alter = a;
   geschlecht = g;
}
```

Sofern Sie ein `static`-Element im `public`-Bereich deklariert haben, ist es kein Problem, mit Hilfe von

```
Mensch person;
// Zugriff auf static-Element ...
// ... wenn anzahlMensch public ist ...
cout << Mensch::anzahlMensch << '\n';
// ... oder auch so ...
cout << person.anzahlMensch << '\n';
```

darauf zuzugreifen. Mir persönlich gefällt die Möglichkeit `Mensch::anzahlMensch` besser, da man hier schneller erkennt, dass es sich um eine statische Variable handelt, die unabhängig vom Objekt ist.

Allerdings werden Sie auch in der Praxis relativ selten ein `static`-Element im `public`-Bereich deklarieren, sondern eher mit Zugriffsmethoden auf das Element zugreifen. Um hierbei auf eine vom Objekt unabhängige Variable zuzugreifen, sollte man auch eine vom Objekt unabhängige Methode erstellen (siehe nächsten Abschnitt).

4.4.15 Statische Klassenmethoden

Im letzten Abschnitt wurde eine vom Objekt unabhängige statische Variable erzeugt und verwendet. Um auf den Wert dieser Variablen im `private`-Bereich zuzugreifen, sollte man in der Praxis eine vom Objekt unabhängige Zugriffsmethode verwenden. Eine solche statische Methode realisieren Sie ebenfalls mit dem Schlüsselwort `static`.

Wird die statische Methode außerhalb der Klasse definiert (was im anschließenden Beispiel der Fall ist), so erfolgt das Schlüsselwort `static` nur bei der Deklaration der Methode. In unserem Beispiel sieht das folgendermaßen aus:

```
// mensch.h
...
class Mensch {
private:
   // Eigenschaften der Klasse Mensch
   char *name;
   int len;
   unsigned int alter;
   bool geschlecht; //0 = männlich; 1 = weiblich
   Haustier tier;
   static int anzahlMensch;

public:
   ...
   static int get_anzahlMensch( void );
};
```

Bei der Definition der Zugriffsmethode wird das Schlüsselwort `static` nicht mehr benötigt:

```
// mensch.cpp
...
int Mensch::anzahlMensch = 0;
...
int Mensch::get_anzahlMensch( void ) {
   return anzahlMensch;
}
```

Zwar können Sie diese Zugriffsmethode jetzt auch noch über jedes Objekt aufrufen, aber man sollte den direkten Aufruf über eine Klasse und den Scope-Operator vorziehen, da hierdurch auch der Eindruck entstehen soll, dass diese Methode nicht vom Objekt abhängig ist. Somit können Sie außerhalb der Klasse folgendermaßen die Anzahl von Menschen abfragen:

```
Mensch::get_anzahlMensch()
```

Den Zugriff auf die neu implementierte statische Eigenschaft (`anzahlMensch`) und die statischen Zugriffsmethoden (`get_anzahlMensch()`) soll das folgende Hauptprogramm demonstrieren:

```
// main.cpp
#include "mensch.h"

int main(void) {
   Mensch person1( "Adam", 22, Mensch::MANN,
                   "Hund", "Dackel", "Waldi" );
   Haustier cat( "Katze", "Perser", "Minzi" );
   Mensch person2( "Eva", 19, Mensch::FRAU, cat );
   Mensch person3;

   cout << "Menschen: "
        << Mensch::get_anzahlMensch() << '\n';
   {
        Mensch person4;
        Mensch person5;
        cout << "Menschen: "
             << Mensch::get_anzahlMensch() << '\n';
   }
   cout << "Menschen: "
        << Mensch::get_anzahlMensch() << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
Menschen: 3
Menschen: 5
Unbekannt ist von uns gegangen
Unbekannt ist von uns gegangen
Menschen: 3
```

4.4.16 Friend-Funktionen bzw. friend-Klassen

Globale Funktionen und Methoden anderer Klassen dürfen normalerweise nicht auf die privaten Elemente einer Klasse zugreifen. Nur so ist sichergestellt, dass Zugriffe auf Objekte von fremden Objekten und Funktionen verhindert werden.

Mit dem Schlüsselwort `friend` können Sie diese Datenkapselung »aufbrechen«. Wenn Sie in einer Klasse eine Funktion oder andere Klassen zum »Freund« (`friend`) erklären, hat dieser »Freund« Zugriff auf alle privaten Elemente. Natürlich bedeutet dies zum Beispiel bei Klassen nicht, dass, wenn eine Klasse eine andere Klasse zum »Freund« erklärt, diese Klasse ebenfalls Zugriff auf die priva-

ten Elemente der anderen Klasse erhält – oder einfacher ausgedrückt, die »Freundschaft« von Klassen beruht nicht auf Gegenseitigkeit. Sonst könnte man jede Datenkapselung beliebig unterlaufen.

Hierzu ein einfaches Beispiel aus unserer Klasse `Mensch`. Ausnahmsweise beginnen wir mal mit dem Hauptprogramm:

```
// main.cpp
#include "mensch.h"
using namespace std;
void print_name( Mensch& p ){
   cout << "Der Name: " << p.name << '\n';
}

int main(void) {
   Mensch person1( "Adam", 22, Mensch::MANN,
                   "Hund", "Dackel", "Waldi" );
   print_name(person1);
   return 0;
}
```

Ohne irgendwelche Vorkehrungen würde dieses Beispiel nicht funktionieren, da hier in einer globalen Funktion, `print_name()`, versucht wird, auf eine private Eigenschaft (`name`) zuzugreifen. Dies ist in der Regel den Klassenmethoden vorbehalten.

Wollen Sie aber jetzt dieser globalen Funktion trotzdem Zugriff auf die `private`-Eigenschaften der Klasse `Mensch` gewähren, brauchen Sie diese Funktion nur in der entsprechenden Klasse mit dem Schlüsselwort `friend` zu deklarieren:

```
// mensch.h
...
class Mensch {
private:
   // Eigenschaften der Klasse Mensch
   char *name;
   int len;
   unsigned int alter;
   bool geschlecht; //0 = männlich; 1 = weiblich
   Haustier tier;
   static int anzahlMensch;
   void test_geschlecht( void );

public:
   friend void print_name( Mensch& MenschPtr);
   ...
};
```

Zwar spielt es keine Rolle, ob Sie diese Deklaration im `public`- oder `private`-Bereich vereinbaren, aber im Grunde soll eine solche Erweiterung als öffentliche Schnittstelle dienen. Deshalb sollte die Vereinbarung sinnvollerweise im `public`-Bereich vorgenommen werden. Mehr ist jetzt nicht nötig, um mit der globalen Funktion `print_name()` auf private Elemente einer Klasse zuzugreifen.

Hinweis [«]

Eine `friend`-Funktion stellt keine Methode einer Klasse dar, weshalb Ihnen auch kein `this`-Zeiger zur Verfügung steht.

Natürlich sind neben `friend`-Funktionen auch `friend`-Klassen möglich. Auch hierbei gilt, dass nur einseitige freundschaftliche Verhältnisse eingegangen werden können. Ein Beispiel einer solchen `friend`-Vereinbarung:

```
// Vorwärtsdeklaration
class B;

class A {
   private:
   int iwert;
   ...
   public:
   friend class B;
   ...
};

class B {
   private:
   ...
   public:
   void func_of_B( void );
}

// Definition
void func_of_B( void ) {
   A a;
   // Möglich, weil Klasse B ein Freund von A ist
   a.iwert = 100;
}
```

Eingesetzt wird die `friend`-Technik häufig, wenn man globale Funktionen verwenden will und keine Klassenmethoden geeignet sind. Als Beispiel dient hier das Überladen von Operatoren.

Allerdings sollte man immer beachten, dass Sie mit der `friend`-Deklaration das Prinzip der Datenkapselung der OOP aufweichen. Daher sollte man es nur einsetzen, wenn es nicht anders möglich ist. Gerade in der Praxis hat sich schon oft gezeigt, dass der Zugriff von »fremden« Funktionen auf private Daten einer Klasse zu Fehlern führt. Dies muss nicht zwangsläufig der Fall sein, aber spätestens, wenn Sie eine Klasse ändern oder erweitern und dabei nicht aufpassen, treten Probleme auf.

Besonders die Verwendung von `friend`-Klassen gilt als schlechtes Design. In solch einem Fall sollte man den Entwurf der Klasse nochmals überprüfen.

4.4.17 Zeiger auf Eigenschaften einer Klasse

Neben den üblichen Zeigern, mit denen Sie bestimmte Variablen adressieren können, können Sie in C++ auch Zeiger verwenden, die auf bestimmte Elemente eines Objekts zeigen können. Dabei ist sowohl die Adressierung von Eigenschaften also auch von Methoden möglich. Die Deklaration von solchen Zeigern sieht im Gegensatz zu herkömmlichen Zeigern wie folgt aus:

```
// Herkömmliche Syntax
Typ* Ptr;
// Elementzeiger
Typ Klasse::*ElementPtr;
```

Es wird gleich deutlich, dass dieser Zeiger nur auf Elemente einer bestimmten Klasse verweisen kann. Natürlich gelten hier nach wie vor die üblichen Zugriffsrechte. Von außen können Sie aber nicht auf `private`-Eigenschaften oder Methoden einer Klasse zugreifen. Für solche Fälle müssen Sie entweder Methoden oder `friend`-Funktionen (siehe Abschnitt 4.4.16, »Friend-Funktionen bzw. friend-Klassen«) verwenden. Neben der Klasse sind solche Elementzeiger auch an den Datentyp gebunden, den diese referenzieren.

Außerdem gibt es einen weiteren »internen« Unterschied zwischen Elementzeigern und herkömmlichen Zeigern. Denn im Grunde hat jedes Objekt eine Anfangsadresse, und die Adresse auf das Element, auf das der Elementzeiger verweist, stellt das Offset dar. Wenn Sie zum Beispiel ein ganzes Array von Objekten haben, zeigt ein einmal referenzierter Zeiger daher immer auf dasselbe Element eines jeden Klassen-Arrays – das heißt, es muss nicht immer wieder von neuem referenziert werden.

Vor dem Zugriff auf den Elementzeiger müssen Sie diesen selbstverständlich auch initialisieren. Hierbei kommt wieder die übliche Variante mit Hilfe des Adressoperators zum Einsatz:

```
ElementPtr = &Klassenname::Element;
```

Da hierbei auf einzelne Elemente einer Klasse zugegriffen wird, gibt es mit `.*` und `->*` zwei neue Operatoren. Wobei der Zugriff analog zu den Gegenstücken des Punkt- und Pfeiloperators funktioniert. Beispielsweise wird der Operator `.*` zum Zugriff über ein Objekt wie folgt verwendet:

```
Objekt.*ElementPtr
```

Der Operator `->*` wird hingegen zum Zugriff über einen Zeiger auf das Objekt folgendermaßen verwendet:

```
ObjektPtr->*ElementPtr
```

Der Elementzeiger soll bei der Klasse `Mensch` verwendet werden. Um hier keinen allzu großen Aufwand zu betreiben, soll eine globale `friend`-Funktion in das Beispiel eingebaut werden. Fügen Sie daher in der Header-Datei *mensch.h* folgende Deklaration der Funktion `durchschnitt()` in den `public`-Bereich ein:

```
// mensch.h
...
class Mensch {
private:
   ...

public:
   friend int durchschnitt( Mensch* MenschPtr, int anzahl );
};
```

Diese Funktion soll den Altersdurchschnitt mehrerer erzeugter Menschen berechnen. Hierzu benötigen Sie nur noch die Hauptfunktion mit der globalen Funktion `durchschnitt()`:

```
// main.cpp
#include "mensch.h"

int durchschnitt( Mensch* MenschPtr, int anzahl ) {
   unsigned int gesamt = 0;
   unsigned int Mensch::*alterPtr;
   // alterPtr zeigt immer auf das Element alter in der
   // Klasse Mensch
   alterPtr = &Mensch::alter;
   for( int i=0; i<anzahl; i++ ) {
      gesamt += MenschPtr[i].*alterPtr;
   }
   return (gesamt/anzahl);
}
```

```
int main(void) {
   Mensch personen[] = {
      Mensch( "Adam", 22, Mensch::MANN ),
      Mensch( "Eva", 19 ),
      Mensch( "Jürgen", 30, Mensch::MANN  ),
      Mensch( "Jonathan", 3, Mensch::MANN ),
      Mensch( "Fatma", 36 )
   };
   int alter_schnitt = durchschnitt(
      personen, (sizeof(personen)/sizeof(Mensch)) ) ;
   cout << "Durchschnittsalter der "
        << (sizeof(personen)/sizeof(Mensch))
        << " Personen ist : " << alter_schnitt << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
Durchschnittsalter der 5 Personen ist : 22
```

Natürlich soll das Beispiel nicht darüber hinwegtäuschen, dass hier über den Zeiger nur ein lesender Zugriff auf die Klassenelemente möglich ist. Sie können hiermit auch auf jeden anderen Typ der Klasse zugreifen, wenn dieser den Angaben entspricht:

```
unsigned int Mensch::*alterPtr;
```

Würden sich in der Klasse `Mensch` weitere Elemente mit `unsigned int` befinden, so könnten Sie diesen Zeiger auch ohne Probleme für diese Elemente verwenden. Außerdem kann hiermit ein Zeiger auf Methoden vereinbart werden, was aber den Code unnötig verkomplizieren kann.

Das Beispiel mit der Klasse `Mensch` demonstriert den Elementzeiger allerdings recht ungenau. Daher hierzu nochmals ein unabhängiges Beispiel, das auch die Verwendung über einen Zeiger auf ein Objekt, den Operator `->*`, demonstriert:

```
// elementzeiger.cpp
#include <iostream>
using namespace std;

class Elementzeiger {
public:
   int awert;
   int bwert;
   // Konstruktor
   Elementzeiger (int a, int b) {
      awert = a;
```

```
      bwert = b;
   }
   // Implizit inline
   void print (void) {
      cout << "\t awert ist " << awert << '\n';
      cout << "\t bwert ist " << bwert << '\n';
   }
};

int main (void) {
   Elementzeiger objekt1 (1,2), objekt2 (3,4);
   Elementzeiger *Ptrobjekt;
   cout << "Werte von objekt1:\n";
   objekt1.print ();
   cout << "Werte von objekt2:\n";
   objekt2.print ();

   // Zeiger auf int Elemente in der Klasse Elementzeiger
   int (Elementzeiger::*IntegerElementzeiger);

   // Setzen des Elementzeigers auf Element awert
   IntegerElementzeiger = &Elementzeiger::awert;
   // Zugriff Elementzeiger: awert in objekt1 wird geaendert
   objekt1.*IntegerElementzeiger = 11;
   // Zugriff Elementzeiger: awert in objekt2 wird geaendert
   objekt2.*IntegerElementzeiger = 33;

   // Setzen des Elementzeigers auf Element bwert
   IntegerElementzeiger = &Elementzeiger::bwert;

   // Zugriff Elementzeiger: bwert in objekt1 wird geaendert
   objekt1.*IntegerElementzeiger = 22;
   // Zugriff Elementzeiger: bwert in objekt2 wird geaendert
   objekt2.*IntegerElementzeiger = 44;

   cout << "\nWerte von objekt1:\n";
   objekt1.print ();
   cout << "Werte von objekt2:\n";
   objekt2.print ();

   // Zugriff wie oben (auf bwert der beiden Objekte)
   // nur über Zeiger auf ein Elementzeiger-Objekt
   Ptrobjekt = &objekt1;
   Ptrobjekt->*IntegerElementzeiger = 222;
   Ptrobjekt = &objekt2;
   Ptrobjekt->*IntegerElementzeiger = 444;
```

```
   cout << "\nWerte von objekt1:\n";
   objekt1.print ();
   cout << "Werte von objekt2:\n";
   objekt2.print ();
   return 0;
}
```

Das Programm bei der Ausführung:

```
Werte von objekt1:
         awert ist 1
         bwert ist 2
Werte von objekt2:
         awert ist 3
         bwert ist 4

Werte von objekt1:
         awert ist 11
         bwert ist 22
Werte von objekt2:
         awert ist 33
         bwert ist 44

Werte von objekt1:
         awert ist 11
         bwert ist 222
Werte von objekt2:
         awert ist 33
         bwert ist 444
```

In der Praxis sind allerdings genau diese Beispiele weniger tauglich, da hierbei immer wieder die Datenkapselung unterlaufen wird. Einmal wurde eine globale `friend`-Funktion verwendet, und das zweite Mal wurden die Eigenschaften einfach `public` gemacht. Dass dies nicht im Sinne der OOP ist, wurde bereits erwähnt. Wollen Sie dennoch diesen Vorgang mit dem Elementzeiger realisieren, sollten Sie ihn auf die Zugriffsmethoden anwenden.

Im Folgenden wurde das Beispiel *elementzeiger.cpp* um weitere Zugriffsmethoden erweitert, die im Code mit Elementzeigern verwendet werden sollen:

```
// elementzeiger2.cpp
#include <iostream>
using namespace std;

class Elementzeiger {
   private:
   int awert;
```

```
   int bwert;
   public:
   // Konstruktor
   Elementzeiger (int a, int b) {
      awert = a;
      bwert = b;
   }
   void print (void) {
      cout << "\t awert ist " << awert << "\n";
      cout << "\t bwert ist " << bwert << "\n";
   }
   int get_awert( void ) { return awert; }
   int get_bwert( void ) { return bwert; }
   void set_awert( int a ) { awert = a; }
   void set_bwert( int b ) { bwert = b; }
};

int main (void) {
   Elementzeiger objekt1 (1,2), objekt2 (3,4);
   // Zeiger auf Methode void (int) in Klasse Elementzeiger
   void (Elementzeiger::*VoidMethodeIntParamPtr)(int);
   // Zeiger auf Methode int (void) in Klasse Elementzeiger
   int (Elementzeiger::*IntMethodeVoidParamPtr)(void);

   // Zeiger auf die Methode get_awert setzen
   IntMethodeVoidParamPtr = &Elementzeiger::get_awert;
   // Zeiger auf die Methode set_awert setzen
   VoidMethodeIntParamPtr = &Elementzeiger::set_awert;

   // set_awert für objekt1 ausführen
   (objekt1.*VoidMethodeIntParamPtr)(11);
   // set_awert für objekt2 ausführen
   (objekt2.*VoidMethodeIntParamPtr)(33);

   // Aufruf von get_awert für beide Objekte
   cout << "awert (objekt1): "
        <<   (objekt1.*IntMethodeVoidParamPtr)() << "\n";
   cout << "awert (objekt2): "
        <<   (objekt2.*IntMethodeVoidParamPtr)() << "\n";

   // Selbiger Vorgang jetzt für bwert der beiden Objekte

   // Zeiger auf die Methode get_bwert setzen
   IntMethodeVoidParamPtr = &Elementzeiger::get_bwert;
   // Zeiger auf die Methode set_bwert setzen
   VoidMethodeIntParamPtr = &Elementzeiger::set_bwert;
```

```
   // set_bwert für objekt1 ausführen
   (objekt1.*VoidMethodeIntParamPtr)(22);
   // set_bwert für objekt2 ausführen
   (objekt2.*VoidMethodeIntParamPtr)(44);
   // Aufruf von get_bwert für beide Objekte
   cout << "bwert (objekt1): "
        <<   (objekt1.*IntMethodeVoidParamPtr)() << "\n";
   cout << "bwert (objekt2): "
        <<   (objekt2.*IntMethodeVoidParamPtr)() << "\n";
   return 0;
}
```

Das Programm bei der Ausführung:

```
awert (objekt1): 11
awert (objekt2): 33
bwert (objekt1): 22
bwert (objekt2): 44
```

4.5 Operatoren überladen

Da Sie in C++ mit Klassen bzw. Strukturen neue bzw. fortgeschrittene Typen erzeugen, ist es glücklicherweise auch möglich, die meisten vordefinierten Operatoren zu überladen. So lässt sich zum Beispiel der Operator + so überladen, dass hiermit komplizierte Berechnungen mit komplexen Zahlen oder auch Manipulationen mit Strings gemacht werden können. Vor allem ist es dadurch auch möglich, mit << die neuen Typen auszugeben, mit >> einzulesen und mit = eine Zuweisung vorzunehmen.

4.5.1 Grundlegendes zur Operator-Überladung

Dass in C++ Operatoren überladen werden können, verdanken sie den sogenannten Operatorfunktionen, die implementiert sind. Dies sind Funktionen, die mit dem Schlüsselwort `operator` und dem eigentlichen Operatorsymbol vom Compiler aufgerufen werden. Die meisten Operatoren können überladen werden (siehe Tabelle 4.1 und 4.2).

Operatoren	Bedeutung
+ – * / % - (unär) ++ -- += -= *= /= %=	arithmetische Operatoren
&& || !	logische Operatoren

Tabelle 4.1 Operatoren, die überladen werden können

Operatoren	Bedeutung
== != < <= > >=	Vergleichsoperatoren
=	Zuweisungsoperatoren
& | ^ ~ >> << &= |= ^= >>= <<=	Bit-Operatoren
* -> ->* , () [] & new delete new[] delete[]	sonstige Operatoren

Tabelle 4.1 Operatoren, die überladen werden können (Forts.)

Operatoren	Bedeutung
.	Zugriffsoperator
.*	Zugriffsoperator (Elementzeiger)
::	Scope-Operator (Bereichsoperator)
?:	ternärer Operator (bedingte Auswertung)
sizeof	Größe von Objekten

Tabelle 4.2 Operatoren, die nicht überladen werden können

Schlüsselwort »operator«

Alle Operatoren, die in Tabelle 4.1 aufgelistet wurden, können Sie entweder in der üblichen Operatornotation verwenden oder aber in der Funktionsnotation. Beispielsweise enthält der Ausdruck

```
objekt1 + objekt2
```

den Operator +, mit dem hier zwei Werte addiert werden. Im Fall von numerischen Werten ist das sehr sinnvoll. Aber es wird hierbei von Objekten eines bestimmten Typs ausgegangen. Somit entspricht dieser gezeigte Ausdruck folgender Funktionsnotation:

```
objekt1.operator+(objekt2)
```

Auf diese Weise werden überladene Operatoren wie normale Funktionen definiert – wobei die Syntax eines solchen Funktionsnamens wie folgt auszusehen hat:

```
Klassenname operator@( Parameter )
```

Das Zeichen @ hinter dem Schlüsselwort `operator` muss durch einen Operator ersetzt werden, der natürlich auch überladbar sein muss. Das ist in der Praxis einfacher, als es sich in der Theorie anhört.

Die Regeln

Mit der Operator-Überladung können Sie manches »verbiegen«, aber trotzdem gibt es einige Aspekte, die Sie hierbei beachten müssen oder nicht »verbiegen« können:

- Operator-Überladungen finden immer im Zusammenhang mit Klassen statt. Das heißt, es ist nicht möglich, diese Operatoren bei den Basisdatentypen zu »verbiegen«.
- Es lassen sich keine neuen Operatoren damit erzeugen, das heißt, Sie können nur Operatoren überladen, die bereits existieren. Neue Operatorsymbole (wie z. B. `**`) lassen sich hiermit leider nicht einführen.
- Die Operanden eines Operators können nicht verändert werden. Ein binärer Operator hat nach wie vor zwei Operanden und ein unärer einen. Der ternäre Operator `?:` enfällt hier, da dieser nicht überladen werden kann.
- Auch die Priorität der Operatoren verändert sich nicht. Der Operator `*` zum Beispiel besitzt immer noch eine höhere Priorität als der Operator `+` (Punkt-vor-Strich-Regelung, siehe auch die Prioritätstabelle im Anhang des Buches).
- Operatoren dürfen außerdem keine Default-Argumente erhalten und müssen natürlich dieselbe Argumentenzahl wie der ursprüngliche Operator enthalten.

Beispiele zu Demonstrationszwecken

Es ist schwierig, hierfür einfache Beispiele zu Demonstrationszwecken zu erstellen. Manchmal macht eine Operator-Überladung bei einem Beispiel keinen Sinn, daher habe ich mich entschlossen, zwei verschiedene Themen zu behandeln. Ein Beispiel verwendet eine Klasse mit komplexen Zahlen und ein anderes Beispiel eine Klasse mit Strings (ein Klasse, mit der dynamisch Strings angelegt werden). Beide Beispiele verwenden dabei nichts, was nicht bereits in diesem Buch besprochen wurde. Wenn Sie etwas nicht verstehen, sollten Sie nochmals zum entsprechenden Abschnitt zurückblättern. Die Beispiele werden außerdem in einer Datei zusammengefasst, was in der Praxis natürlich nicht so sein sollte.

[»] **Hinweis**

Natürlich stellt diese `String`-Klasse nur eine Demonstration dar und sollte nicht durch die bereits kurz erwähnte Standard-String-Bibliothek ersetzt werden.

Zuerst das Beispiel mit der dynamischen `String`-Klasse (zum Testen auch immer gleich mit einer Hauptfunktion):

```
// String.cpp
#include <iostream>
#include <cstring>
using namespace std;

class String {
private:
   char *buffer;
   unsigned int len;
public:
   // Konstruktoren
   String( const char* s="") {
      len = strlen(s);
      buffer = new char [len+1];
      strcpy( buffer, s );
   }
   // Destruktor
   ~String() { delete [] buffer; }
   // Kopierkonstruktor - explizit nötig da
   // dynamischer Speicherbereich verwendet wird
   String( const String& s ) {
      len = s.len;
      buffer = new char [len+1];
      strcpy( buffer, s.buffer );
   }
   // Zugriffsmethode
   char* get_String() const { return buffer; }
};

int main( void ) {
   String string1("Adam");
   cout << string1.get_String() << "\n";
   return 0;
}
```

Das zweite Beispiel, das in diesem Abschnitt der Operator-Überladung zur Demonstration verwendet werden soll, benutzt komplexe Zahlen mit einem realen und einem imaginären Anteil. Hier die Klasse `myComplex` (ebenfalls mit einer Hauptfunktion zum Testen):

Hinweis **[«]**

Auch hier gilt, die selbstdefinierte Klasse `myComplex` sollte nicht durch die bereits bewährte Klasse der entsprechenden Standardbibliothek ersetzt werden.

```
// complex.cpp
#include <iostream>
using namespace std;

class myComplex {
private:
   double _real;
   double _image;

public:
   myComplex( double val1=0.0, double val2=0.0 ) {
      _real = val1; _image = val2;
   }
   void print_Complex( ) {
      cout << _real << "+" << _image << "\n";
   }
};

int main( void ) {
   myComplex val1(1.1, 2.2);
   myComplex val2(3.3, 4.4);
   cout << "Wert von val1: "; val1.print_Complex( );
   cout << "Wert von val2: "; val2.print_Complex( );
   return 0;
}
```

Auch hier haben wir uns zunächst auf das Nötigste beschränkt. Mit den beiden Klassen werden Sie in den nächsten Abschnitten den ein oder anderen Operator überladen.

Das Beispiel mit den komplexen Zahlen stellt eine Erweiterung der reellen Zahlen dar. Damit sind komplexere Berechnungen möglich, die mit reellen Zahlen nicht machbar sind. Solche Zahlen werden vorwiegend bei Berechnungen von elektronischen Schaltungen verwendet. Jede dieser Zahlen besitzt einen reellen Anteil (`_real`), den Realteil, der eine reelle Zahl ist, und einen Imaginärteil (`_image`), der ein reelles Vielfaches von i=(Wurzel)$^{-1}$ ist. Im Grunde ist dies allerdings nicht so wichtig, wenn Sie sich nicht mit dieser Mathematik auseinandersetzen wollen – im Beispiel geht es nur um die Operator-Überladung von Klassen.

4.5.2 Überladen von arithmetischen Operatoren

Zunächst soll mit dem Operator += ein einfacher arithmetischer Operator überladen werden. Beispielsweise soll für die Klasse `String` der Ausdruck

```
string1+=string2;
```

dazu führen, dass der Inhalt von `string2` am Ende von `string1` hinzugefügt wird. Beachten Sie bitte, dass + und += zwei verschiedene Operatoren sind.

Operator-Überladung als Klassenmethode

Um also den Inhalt zweier Strings zu »addieren« (sofern man davon sprechen kann), benötigt man zunächst den Rückgabetyp des überladenen Operators. Da Sie hier vorhaben, zwei Objekte der Klasse `String` zu addieren, ist es logisch, dass der Rückgabewert vom Typ `String` ist. Somit könnte die Deklaration dieser Operator-Überladung wie folgt aussehen:

```
String operator+=( const String& str1 );
```

In unserem Fall entspricht die Deklaration der Definition, weil aus Gründen der Übersichtlichkeit gleich alles in eine Quelldatei geschrieben wird. Bevor Sie die komplette Definition und das komplette Listing in Aktion sehen, stellt sich noch die Frage, wie man diese (Operator-Überladungs-)Methode aufrufen kann. Betrachtet man die Syntax der Operator-Überladung, wäre folgender Aufruf möglich:

```
string1.operator+=(string2);
```

Dieser Aufruf der Operatorfunktion mit `operator+=()` hat dieselbe Bedeutung wie

```
string1 += string2;
```

Sie sehen, dass man mit der Operator-Überladung hervorragende Schnittstellen für die Klassen anbieten kann. Allerdings sollte man eine solche Operator-Überladung auch sinnvoll einsetzen. Zum Beispiel ist es in der Praxis sinnlos, zwei Strings zu multiplizieren (`string1 *= string2`).

Hierzu das Beispiel der Operator-Überladung des +=-Operators der Klasse `String`:

```
// String2.cpp
#include <iostream>
#include <cstring>
using namespace std;

class String {
private:
   char *buffer;
   unsigned int len;

public:
   // Konstruktoren
   String( const char* s="") {
      len = strlen(s);
      buffer = new char [len+1];
```

```
      strcpy( buffer, s );
   }
   // Destruktor
   ~String() { delete [] buffer; }
   // Kopierkonstruktor - explizit nötig da
   // dynamischer Speicherbereich verwendet wird
   String( const String& s ) {
      len = s.len;
      buffer = new char [len+1];
      strcpy( buffer, s.buffer );
   }
   // Zugriffsmethode
   char* get_String() const { return buffer; }
   // Definition der Operator-Überladung
   String operator+=( const String& str1 ) {
      // tmp gleich mit aktuellem Objekt initialisieren
      String tmp(*this);
      // Aktuelles Objekt löschen
      delete [] buffer;
      // Gesamtlänge beider Objekte ermitteln
      len = tmp.len + str1.len;
      // Speicher reservieren
      buffer = new char[len]+1;
      // linker Operand in buffer kopieren
      strcpy( buffer, tmp.buffer );
      // rechten Operand anhängen
      strcat( buffer, str1.buffer );
      // Zurück und fertig
      return buffer;
   }
};

int main( void ) {
   String string1("Adam");
   String string2("Eva");
   string1+=" und ";
   string1.operator+=(string2);
   cout << string1.get_String() << "\n";
   return 0;
}
```

Das Programm bei der Ausführung:

```
Adam und Eva
```

Operator-Überladung als globale (»friend«-)Funktion

Die Operator-Überladung lässt sich natürlich auch als normale globale Funktion verwenden. Allerdings werden Sie wegen der Datenkapselung kaum Zugriff auf die Eigenschaften (Daten) der Klasse haben, weil diese gewöhnlich als `private` vereinbart sind. Daher wird eine solche Funktion im Allgemeinen als `friend` deklariert.

Wenn Sie eine Operatorfunktion als globale `friend`-Funktion definieren, müssen Sie jedem Operanden des Operators einen eigenen Parameter widmen. Bei unären Operatoren wäre dies immer ein Operand und bei binären Operatoren zwei Operanden.

Im Beispiel der Klasse `myComplex` wollen wir nun den Operator + mit einer globalen `friend`-Funktion überladen. Zur Demonstration wurde zum Vergleich eine Klassenmethode zur Überladung des Minusoperators verwendet, die, wie schon beim Beispiel der Klasse `String` gesehen, nur einen Parameter benötigt.

Somit sieht die Deklaration der globalen Überladungsfunktion im `public`-Bereich der Klasse `myComplex` wie folgt aus:

```
// Operator-Überladung als globale friend-Funktion
friend myComplex operator+(myComplex val1, myComplex val2);
```

Ansonsten können Sie diese Funktion wie gehabt mit

```
complexSum = complex1 + complex2;
```

aufrufen. Diese Anweisung ist gleichwertig mit dem Funktionsaufruf

```
complexSum = operator+( complex1, complex2 );
```

Hierzu nun das komplette Beispiel, das neben der Überladung des Operators + als globale `friend`-Funktion auch den Minusoperator als Klassenmethode zum Vergleich demonstriert.

```
// complex2.cpp
#include <iostream>
using namespace std;

class myComplex {
private:
   double _real;
   double _image;

public:
   myComplex( double val1=0.0, double val2=0.0 ) {
      _real = val1; _image = val2;
```

```
    }
    void print_Complex( ) {
       cout << _real << "+" << _image << "\n";
    }
    // Operator-Überladung als globale friend-Funktion
    friend myComplex operator+( myComplex v1, myComplex v2 );
    // Operator-Überladung als Klassenmethode
    myComplex operator-( myComplex val2 ) const {

       // tmp gleich mit aktuellem Objekt initialisieren
       myComplex tmp(*this);
       tmp._real -= val2._real;
       tmp._image -= val2._image;
       return tmp;
    }
};

myComplex operator+( myComplex val1, myComplex val2 ) {
    myComplex tmp;
    tmp._real = val1._real += val2._real;
    tmp._image = val1._image += val2._image;
    return tmp;
}

int main( void ) {
    myComplex val1(1.1, 2.2);
    myComplex val2(3.3, 4.4);
    myComplex sum;
    cout << "Wert von val1: "; val1.print_Complex( );
    cout << "Wert von val2: "; val2.print_Complex( );
    sum = val1 + val2;
    cout << "Wert von sum; "; sum.print_Complex( );
    sum = val1 - val2;
    cout << "Wert von sum; "; sum.print_Complex( );
    return 0;
}
```

Das Programm bei der Ausführung:

```
Wert von val1: 1.1+2.2
Wert von val2: 3.3+4.4

Wert von sum; 4.4+6.6
Wert von sum; -2.2+-2.2
```

Vergleich der Implementierungen der Operator-Überladung

Jetzt haben Sie zwei mögliche Implementierungen der Operator-Überladung kennengelernt und werden sich wohl fragen, welche der beiden Varianten die bessere ist. Im Grunde bleibt es dem Programmierer überlassen, wie dieser die Operator-Überladung implementieren will.

Dennoch gibt es Fälle, in denen es keine andere Möglichkeit gibt, da Klassenmethoden als linkes Argument immer ein Objekt ihrer Klasse haben müssen. Schließlich können Methoden auch nur an Objekte einer zugehörigen Klasse versandt werden. Wenn dies nicht mehr möglich ist, müssen Sie eine Operator-Überladung als globale `friend`-Funktion implementieren.

Beispielsweise sei bei der `String`-Klasse Folgendes gegeben:

```
String string1("Adam");
char name[30];
cout << "Bitte Name eingeben: ";
cin >> name;
if( name == string1 ) {
    cout << "Beide Namen sind identisch\n";
}
else {
   cout << "Beide Namen sind unterschiedlich\n";
}
```

In diesem Beispiel wird der ==-Operator (Vergleichsoperator) verwendet, um die Parameter `char*` und `String` zu vergleichen. Es wird hier eine Zeichenkette mit einem Objekt verglichen; daher ist es nicht möglich, eine Klassenmethode als Operator-Überladung zu implementieren, weil die Methode == ein `char*`-Objekt schicken würde. Da man mit Basisdatentypen keine Operatoren überladen kann, müssen Sie diese Funktion als globale `friend`-Funktion implementieren.

Die Deklaration dieser globalen `friend`-Funktion im `public`-Bereich der Klasse `String` sieht demnach so aus:

```
friend bool operator ==( const char* s1, const String s2 );
```

Natürlich keine Theorie ohne Praxis, daher das zuvor Beschriebene als Code:

```
// String3.cpp
#include <iostream>
#include <cstring>
using namespace std;

class String {
private:
```

```
   char *buffer;
   unsigned int len;

public:
   // Konstruktoren
   String( const char* s="") {
      len = strlen(s);
      buffer = new char [len+1];
      strcpy( buffer, s );
   }
   // Destruktor
   ~String() { delete [] buffer; }
   // Kopierkonstruktor - explizit nötig, da
   // dynamischer Speicherbereich verwendet wird
   String( const String& s ) {
      len = s.len;
      buffer = new char [len+1];
      strcpy( buffer, s.buffer );
   }
   // Zugriffsmethode
   char* get_String() const { return buffer; }
   friend bool operator ==(const char* s1,const String s2);
};

bool operator ==( const char* s1, const String s2 ) {
   if( strlen(s1) == 0 ) { return false; }
   if( s2.len == 0 )     { return false; }
   if( strcmp(s1, s2.buffer ) == 0 ) {
      return true;
   }
   return false;
}

int main( void ) {
   String string1("Adam");
   char name[30];

   cout << "Bitte Name eingeben: ";
   cin >> name;

   if( name == string1 ) {
      cout << "Beide Namen sind identisch\n";
   }
   else {
      cout << "Beide Namen sind unterschiedlich\n";
```

```
   }
   return 0;
}
```

Da die Operator-Elementfunktion im Kontext des linken Operanden aufgerufen wird (das heißt, das linke Argument einer Klassenmethode muss ein Objekt der eigenen Klasse sein), ist es möglich, dass sich Folgendes wiederum als Klassenmethode implementieren ließe:

```
String string1("Adam");
char name[30];
cout << "Bitte Name eingeben: ";
cin >> name;
// Vertauscht !!!!
if( string1 == name ) {
   cout << "Beide Namen sind identisch\n";
}
else {
   cout << "Beide Namen sind unterschiedlich\n";
}
```

Beide Seiten sind jetzt vertauscht, und der Methode == wird ein Objekt `String` geschickt. Hier wird also (umgekehrt) ein String (Klasse) mit einer Zeichenkette (`char*`) verglichen. Die Klassenmethode hierzu will ich Ihnen nicht vorenthalten:

```
bool operator ==( const char *s1 ) {
   String tmp(*this);
   if( strlen(s1) == 0 ) { return false; }
   if( tmp.len == 0 )     { return false; }
   if( strcmp(s1, tmp.buffer ) == 0 ) {
      return true;
   }
   return false;
}
```

Dasselbe Problem ist auch bei der Klasse `myComplex` gegeben, wenn Sie versuchen, einen `double`-Wert zu einem `myComplex`-Wert zu addieren:

```
ComplexSum = 3.3 + complex1;
```

Auch hier ist das linke Argument des Operators + kein Objekt der Klasse `myComplex`, sondern lediglich ein Basisdatentyp (`double`). Daher müssen Sie eine globale `friend`-Funktion verwenden, die Sie wie folgt deklarieren können:

```
friend myComplex operator+( double val1, myComplex val2 );
```

Hierzu das komplette Beispiel zur +-Operator-Überladung der Klasse `myComplex`, in dem der linke Operand wieder kein Objekt der Klasse ist.

```
// complex3.cpp
#include <iostream>
using namespace std;

class myComplex {
private:
   double _real;
   double _image;

public:
   myComplex( double val1=0.0, double val2=0.0 ) {
      _real = val1; _image = val2;
   }
   void print_Complex( ) {
      cout << _real << "+" << _image << "\n";
   }
   // Operator-Überladung als globale friend-Funktion
   friend myComplex operator+( myComplex v1, myComplex v2 );
   friend myComplex operator+( double v1, myComplex v2 );
};

myComplex operator+( myComplex val1, myComplex val2 ) {
   myComplex tmp;
   tmp._real = val1._real + val2._real;
   tmp._image = val1._image + val2._image;
   return tmp;
}

myComplex operator+( double val1, myComplex val2 ) {
   myComplex tmp;
   tmp._real = val1 + val2._real;
   tmp._image = val2._image;
   return tmp;
}

int main( void ) {
   myComplex val1(1.1, 2.2);
   myComplex sum;
   sum = 3.3 + val1;
   cout << "Wert von sum; "; sum.print_Complex( );
   return 0;
}
```

Wenn folgender Ausdruck gegeben ist

```
ComplexSum = complex1 + 3.3;
```

kann wieder eine Klassenmethode verwendet werden, weil der linke Operand ein Objekt der entsprechenden Klasse ist. Die Klassenmethode sieht demnach wie folgt aus:

```
myComplex operator+( double val2 ) const {
   // tmp gleich mit aktuellem Objekt initialisieren
   myComplex tmp(*this);
   tmp._real += val2;
   return tmp;
}
```

Hinweis [«]

Es ist nicht bei jedem Operator zulässig, diesen als `friend`-Funktion zu definieren. Die Operatoren `=`, `[]`, `()` und `->` müssen immer als Klassenmethoden implementiert werden! Außerdem ist es für einige binäre Operatoren wie `+=`, `-=`, `*=`, `/=`, `%=`, `>>=` und `<<=` besser, sie als Methode zu implementieren, da diese Operatoren nicht »symmetrisch« sind und als linken Operanden stets ein Objekt benötigen (L-value).

4.5.3 Überladen von unären Operatoren

Wenn Sie unäre Operatoren mit einer Methode überladen, so muss der Operand natürlich auch ein Objekt der Klasse sein. Da eine solche Methode, wie jede andere auch, einen `this`-Zeiger auf das aktuelle Objekt enthält, wird sie ohne Parameter definiert.

Für unsere Klasse `myComplex` soll der Vorzeichenoperator überladen werden, beispielsweise:

```
ComplexVal = -complex1;
// ComplexVal = complex1.operator-()
```

Hiermit verändert der Vorzeichenoperator die Vorzeichen des Real- und des Imaginärteils.

Im Gegensatz zum Minusoperator benötigt der unäre Vorzeichen-(Minus-)Operator nur einen Operanden (`-val`) – der binäre Minusoperator benötigt zwei (`val1-val2`). Natürlich muss auch der Rückgabewert von der Klasse `myComplex` sein. Mit dieser Überladung ist auch folgender Ausdruck der Klasse zulässig:

```
ComplexVal = -(complex1 + complex2);
```

Hierzu das Beispiel, in dem der Vorzeichenoperator überladen wird:

```
// complex4.cpp
#include <iostream>
using namespace std;

class myComplex {
private:
   double _real;
   double _image;

public:
   myComplex( double val1=0.0, double val2=0.0 ) {
      _real = val1; _image = val2;
   }
   void print_Complex( ) {
      cout << _real << " + " << _image << "\n";
   }
   myComplex operator-() const {
      return myComplex( -_real, -_image );
   }
};

int main( void ) {
   myComplex val1(1.1, 2.2);
   myComplex sum = -val1;
   cout << "Wert von sum: "; sum.print_Complex( );
   return 0;
}
```

Das Programm bei der Ausführung:

```
Wert von sum: -1.1 + -2.2
```

Wenn Sie einen unären Operator als `friend`-Funktion implementieren wollen, benötigen Sie einen Parameter, da diese Funktion nicht mehr zur Klasse gehört. Der Parameter gibt dann den Operanden an und ist immer vom Typ einer Klasse, deren Operator überladen werden soll.

Zur Abwechslung soll der Operator `!` mit der Klasse `String` überladen und als globale `friend`-Funktion implementiert werden. Damit soll eine Abfrage wie

```
if(!string1) {
   cout << "string1 ist leer\n";
}
else {
   cout << "string1 ist nicht leer\n";
}
```

Auskunft darüber geben, ob ein String »leer« oder bereits initialisiert ist. Somit sieht die Deklaration dieser `friend`-Funktion wie folgt aus:

```
friend bool operator!( const String s );
```

Hierzu das komplette Listing:

```
// String4.cpp
#include <iostream>
#include <cstring>
using namespace std;

class String {
private:
   char *buffer;
   unsigned int len;

public:
   // Konstruktoren
   String( const char* s="") {
      len = strlen(s);
      buffer = new char [len+1];
      strcpy( buffer, s );
   }
   // Destruktor
   ~String() { delete [] buffer; }
   // Kopierkonstruktor - explizit nötig, da
   // dynamischer Speicherbereich verwendet wird
   String( const String& s ) {
      len = s.len;
      buffer = new char [len+1];
      strcpy( buffer, s.buffer );
   }
   // Zugriffsmethode
   char* get_String() const { return buffer; }
   // Operator ! überladen
   friend bool operator!( const String s );
};

bool operator!( const String s ) {
   if( strcmp(s.buffer, "") == 0)
      return true;
   return false;
}

int main( void ) {
```

```
   String string1;

   if(!string1) {
      cout << "string1 ist leer\n";
   }
   else {
      cout << "string1 ist nicht leer\n";
   }
   return 0;
}
```

[»] **Hinweis**

Der C++-Standard schreibt außerdem vor, dass der NOT-Operator (`!`) immer einen `bool`-Wert zurückliefern muss – also entweder wahr (`true`) oder falsch (`false`).

4.5.4 Überladen von ++ und --

Einen besonderen Fall haben Sie hier mit den unären Operatoren `++` und `--`, die in zwei verschiedenen Schreibweisen auftreten können, und zwar in der Postfix- (`val++`) und in der Präfix-Schreibweise (`++val`). Daher müssen Sie für beide Varianten je eine Klassenmethode (oder `friend`-Funktion) erstellen.

Die Präfix-Methode (`++val`) stellt wohl den einfacheren Fall dar. Wichtig ist hierbei, dass Sie in der Methode zuerst die Addition ausführen und dann das Ergebnis zurückliefern. Des Weiteren müssen Sie (logischerweise) das Objekt selbst verändern und benötigen kein temporäres Objekt. Hierzu die komplette Klassenmethode, wie man die Präfix-Schreibweise des `++`-Operators überladen kann:

```
// Präfix-Operator ++ überladen
myComplex& operator++() {
   _real++;
   _image++;
   return *this;
}
```

Der andere Fall, die Überladung in der Postfix-Schreibweise (`val++`), benötigt einen Parameter – einen sogenannten Dummy-Parameter. Hierbei müssen Sie allerdings die Regeln der Postfix-Schreibweise beachten. Und zwar muss zunächst der Ursprungswert des Objekts zurückgegeben werden, was man mit der Rückgabe eines temporären Objekts »simulieren« kann, da der Postfix-Operator ja die Addition erst nach der Auswertung des Objekts durchführen soll. Hier die Methode zum Überladen des Operators `++` in der Postfix-Schreibweise:

```
// Postfix-Operator ++ überladen
myComplex& operator++(int) {
```

```
   // Für den Rückgabewert
   myComplex tmp(*this);
   // Originalwerte verändern
   _real++;
   _image++;
   // ursprünglichen Wert zurückgeben
   return tmp;
}
```

Das folgende Beispiel demonstriert den Einsatz des unären ++-Operators in der Postfix- und in der Präfix-Schreibweise. Die Überladungen wurden als Methoden implementiert. Analog gilt dasselbe natürlich auch für den Operator --:

```
// complex5.cpp
#include <iostream>
using namespace std;

class myComplex {
private:
   double _real;
   double _image;

public:
   myComplex( double val1=0.0, double val2=0.0 ) {
      _real = val1; _image = val2;
   }
   void print_Complex( ) {
      cout << _real << "+" << _image << "\n";
   }
   // Präfix-Operator ++ überladen
   myComplex& operator++() {
      _real++;
      _image++;
      return *this;
   }
   // Postfix-Operator ++ überladen
   myComplex& operator++(int) {
      static myComplex tmp(*this);
      _real++;
      _image++;
      return tmp;
   }
};

 int main( void ) {
   myComplex val1(1.1, 2.2);
```

```
    ++val1;
    cout << "Wert von val1: "; val1.print_Complex( );
    val1++;
    cout << "Wert von val1: "; val1++.print_Complex( );
    cout << "Wert von val1: "; val1.print_Complex( );
    return 0;
}
```

Das Programm bei der Ausführung:

```
Wert von val1: 2.1+3.2
Wert von val1: 3.1+4.2
Wert von val1: 4.1+5.2
```

4.5.5 Überladen des Zuweisungsoperators

Das Überladen des Zuweisungsoperators = wurde ja bereits in Abschnitt 4.4.8, »Dynamisch erzeugte Objekte kopieren ('operator=()')«, in der Praxis eingesetzt. Verwenden Sie zum Beispiel den Zuweisungsoperator mit komplexen Zahlen wie

```
myComplex val1(1.1, 2.2);
myComplex val2 = val1;
```

so erhält nach dieser Zuweisung `val2` tatsächlich den Inhalt der komplexen Zahl `val1`. Dass dies funktioniert, liegt daran, dass C++ im Standard-Zuweisungsoperator für Klassen ein *Shallow Copy a*usführt. Somit ist im Grunde gar keine Überladung des Zuweisungsoperators nötig, da der von C++ zur Verfügung gestellte Standard-Zuweisungsoperator gute Arbeit verrichtet.

Anders sieht die Sache allerdings bei Klassen aus, die dynamische Eigenschaften enthalten, wie dies beispielsweise bei der Klasse `String` mit `buffer` der Fall ist.

```
String string1("Adam");
String string2 = string1;
```

Wenn Sie das Beispiel testen, werden Sie auf den ersten Blick und bei der ersten Ausgabe keinen Fehler bemerken. Allerdings täuscht dies, denn durch die Zuweisung eines Objekts, das dynamische Eigenschaften enthält, besitzen beide Objekte Zeiger auf den gleichen Speicherbereich. Wird eines der beiden Objekte gelöscht, bleibt hier mindestens ein Zeiger erhalten, der auf einen nicht mehr allozierten Speicherbereich im Heap zeigt. Solche »wilden« Zeiger führen häufig bei Programmen zu einem Absturz.

Somit benötigen Sie also für Objekte mit dynamischen Eigenschaften und bei der Verwendung des Zuweisungsoperators eine »tiefe« Kopie (*Deep Copy*), Sie müssen in diesem Fall den Zuweisungsoperator überladen.

Eine solche Überladung geschieht in mehreren Schritten. Zunächst müssen Sie überprüfen, ob eine Zuweisung wie `objekt=objekt` vorliegt (was zwar keinen Sinn ergibt, aber rein syntaktisch auch kein Fehler ist). Hiermit würden Sie praktisch das Objekt selbst zerstören, was unter Umständen zu einem fehlerhaften Verhalten des Operators führen kann. Anschließend müssen Sie den bereits allozierten Speicherbereich des Objekts freigeben. Jetzt können Sie die Speicherbereiche mit der richtigen Größe neu anfordern. Danach können Sie den dynamisch allozierten Speicherbereich verwenden (kopieren) und abschließend als Rückgabe eine Referenz auf das eigene Objekt (`*this`) zurückgeben.

Hinweis [«]

Eine Regel sollte sein, dass man beim Vorhandensein dynamischer Eigenschaften einer Klasse neben dem Kopierkonstruktor auch immer den Zuweisungsoperator selbst definiert.

Somit sieht die Überladung des Zuweisungsoperators mit der Klasse `String` folgendermaßen aus:

```
String& operator=( const String& s ) {
   // Auf Selbst-Zuweisung überprüfen
   if( this == &s ) {
      return *this;
   }
   delete[] buffer;
   buffer = 0;
   len = s.len;
   buffer = new char[len+1];
   strcpy( buffer, s.buffer );
   return *this;
}
```

Zuweisung von Objekten verhindern

Es ist standardmäßig immer erlaubt, einem Objekt einer Klasse ein anderes Objekt der gleichen Klasse zuzuweisen (auch ohne Überladung des Zuweisungsoperators übernimmt dies der Compiler). Hierbei wird immer Eigenschaft für Eigenschaft kopiert. Für den Fall, dass Sie dieses Verhalten unterbinden wollen, müssen Sie nur das Überladen des Zuweisungsoperators mit einer leeren Methodenfunktion innerhalb eines privaten Bereiches (`private`) definieren. Private Mitglieder sind von außen nicht zugänglich, so kann man außerhalb der Klasse keinem Objekt der Klasse ein Objekt gleicher Klasse mehr zuweisen, weil es eine `private`-Methode ist.

4.5.6 Überladen des Indexoperators »[ ]« (Arrays überladen)

Wenn Sie jetzt in der Klasse `String` Folgendes ausführen wollten

```
String string1 = "Adam";
cout << string1[0] << string1[3] << '\n';
```

würde der Compiler eine Fehlermeldung ausgeben. Schließlich handelt es sich bei `String` um eine Klasse und nicht (direkt) um ein `char`-Array. Wollen Sie also auf einzelne Elemente eines Arrays in einer Klasse zugreifen, müssen Sie den `[]`-Operator überladen. Die Verwendung einer solchen Klasse hat auch den Vorteil, dass man eine Indexprüfung mit einbauen kann. Natürlich hat solch eine Indexüberprüfung immer den Nachteil, dass dies die Laufzeit des Programms negativ beeinflusst. Hier müssen Sie als Programmierer selbst einschätzen können, ob es Ihnen mehr auf die Laufzeit oder auf die Sicherheit ankommt – was sicherlich auch von der Art der Anwendung abhängt, die Sie erstellen.

Da hier einzelne Zeichen verwendet werden, sollten Sie als Rückgabewert eine Referenz auf einen `char`-Typ zurückgeben. Außerdem müssen Sie beim Überladen des Indexoperators in Klassen folgende Punkte beachten:

- Das Überladen des Indexoperators darf nur als Klassenmethode definiert werden – somit ist der linke Operand immer das Objekt der Klasse (**`str`**`[0]`).
- Das rechte Argument wird als Argument an die Funktion übergeben und darf nur einen Parameter haben (`str[`**`0`**`]`).
- Der Typ des rechten Operanden ist beliebig, ebenso wie der Rückgabetyp nicht festgelegt ist.

Mit diesen Freiheiten im Vergleich zu herkömmlichen Arrays können Sie den Indexoperator erheblich verändern. Allerdings sollte man ihn in der Praxis nur so verändern, dass er noch der ursprünglichen Verwendung von Arrays entspricht.

Hierzu das Programmbeispiel, das den Indexoperator überlädt und einzelne Zeichen aus der Klasse `String` zurückgibt und außerdem noch eine Indexüberprüfung vornimmt. Im Falle eines Fehlers (Bereichsüberschreitung) wird ein Fragezeichen zurückgegeben:

```
// String5.cpp
#include <iostream>
#include <cstring>
using namespace std;

class String {
private:
```

```
   char *buffer;
   unsigned int len;

public:
   // Konstruktoren
   String( const char* s="") {
      len = strlen(s);
      buffer = new char [len+1];
      strcpy( buffer, s );
   }
   // Destruktor
   ~String() { delete [] buffer; }
   // Kopierkonstruktor - explizit nötig, da
   // dynamischer Speicherbereich verwendet wird
   String( const String& s ) {
      len = s.len;
      buffer = new char [len+1];
      strcpy( buffer, s.buffer );
   }
   // Zugriffsmethode
   char* get_String() const { return buffer; }
   // Überladung des Indexoperators
   const char& operator[](unsigned int index) const {
       static char q = '?';
       // Indexüberprüfung
       if( (index >= 0) && (index < len) )
          return buffer[index];
       // Im Fehlerfall ein Fragezeichen zurückgeben
       return q;
   }
};

int main( void ) {
   String string1 = "Adam";
   cout << string1[0] << string1[3] << '\n';
   // !!! Bereichsüberschreitung !!!
   cout << string1[99] << "\n";
   return 0;
}
```

Das Programm bei der Ausführung:

```
Am
?
```

Da bei der Überladung des Indexoperators ein beliebiger Datentyp verwendet werden kann, können Sie hiermit auch assoziative Arrays realisieren. Dies sind Arrays, bei denen die einzelnen Elemente über Strings als Index angesprochen werden. Außerdem könnte man mit jedem einzelnen Array-Element Speicher dynamisch reservieren und somit eine ganze Liste von dynamischen Objekten verwalten.

Hierzu ein einfaches Beispiel unserer Klasse `String`, die jetzt auch assoziative Arrays unterstützt. In unserem Fall verwenden wir dies, indem wir im »String« nach einem Teilstring suchen. Den Teilstring geben Sie als Indexwert des Arrays an:

```
// String6.cpp
#include <iostream>
#include <cstring>
using namespace std;

class String {
private:
   char *buffer;
   unsigned int len;

public:
   // Konstruktoren
   String( const char* s="") {
      len = strlen(s);
      buffer = new char [len+1];
      strcpy( buffer, s );
   }
   // Destruktor
   ~String() { delete [] buffer; }
   // Kopierkonstruktor - explizit nötig, da
   // dynamischer Speicherbereich verwendet wird
   String( const String& s ) {
      len = s.len;
      buffer = new char [len+1];
      strcpy( buffer, s.buffer );
   }
   // Zugriffsmethode
   char* get_String() const { return buffer; }
   // Indexoperator überladen
   const char& operator[](unsigned int index) const {
       static char q = '?';
       // Indexüberprüfung
       if( (index >= 0) && (index < len) )
```

```
            return buffer[index];
         // Im Fehlerfall ein Fragezeichen zurückgeben
         return q;
   }

   // Assoziatives Array
   char* operator[] ( const char *str ) const {
      return strstr( buffer, str );
   }
};

int main( void ) {
   String string1 = "Adam und Eva";
   char *ptr = string1["und"];
   if( ptr != NULL )
      cout << ptr << "\n";
   ptr = string1["Ev"];
   if( ptr != NULL )
      cout << ptr << "\n";
   return 0;
}
```

Das Programm bei der Ausführung:

```
und Eva
Eva
```

4.5.7 Shift-Operatoren überladen

Natürlich ist es auch möglich, dass der Eingabeoperator `>>` und der Ausgabeoperator `<<` überladen werden. Damit können Sie eigene Datentypen wie beispielsweise der Klasse `myComplex` in das C++-System implementieren. Dass bisherige Versuche wie

```
cout << complexVal;
```

fehlgeschlagen sind, liegt daran, dass `cout` den Typ des Wertes nicht kennt, das heißt, dass es hierfür keine Ausgabefunktion wie für `char`, `int`, `double` etc. gibt. Für Ein- bzw. Ausgabefunktionen von Klassen sind Sie selbst verantwortlich.

Beginnen wollen wir mit dem Operator `>>` und der Klasse `myComplex`. Mit dem bisherigen Wissen würden Sie zum Einlesen einer komplexen Zahl eine Zugriffsfunktion schreiben. Durch das Überladen des Operators `>>` können Sie hierbei nun ein Objekt der Klasse `myComplex` mit Hilfe von

```
cin >> complexVal;
```

einlesen. Links des Operators steht hier `cin`, was ein Objekt der Klasse `istream` ist. `istream` ist wiederum in der Header-Datei <*iostream*> definiert.

[»] **Hinweis**

Mehr zu den grundlegenden Streams von C++ entnehmen Sie bitte dem Abschnitt 7.2, »Ein-/Ausgabe Klassenhierarchie (I/O-Streams)«.

Da allerdings `cin` ein Objekt der Klasse `istream` ist, haben Sie keinen Zugriff auf die `public`-Eigenschaften (stehen auf der rechten Seite des Operators `>>`) der Klasse `myComplex`. Aus diesem Grund müssen Sie den Operator als globale `friend`-Funktion implementieren. Somit sieht die Syntax der Funktionsdeklaration (bezogen auf die Klasse `myComplex`) folgendermaßen aus:

```
friend istream& operator >>(istream& is, myComplex& val);
```

Als ersten Parameter benötigen Sie eine Referenz der zu definierenden Operatorfunktion (also auf den Eingabe-Stream) – daher auch eine Referenz auf die Klasse `istream`. Der zweite Parameter (der rechte Operand) ist ein Objekt der Klasse `myComplex`, was ebenfalls eine Referenz darauf sein sollte. Der Rückgabewert ist eine Referenz auf `istream`. Dies ist nötig, damit Sie den Operator `>>` mehrmals hintereinander aufrufen können:

```
// Liest zwei komplexe Zahlen ein
cin >> complex1 >> complex2;
```

Somit ist die Anweisung

```
cin >> val;
```

gleichwertig zu

```
operator>>(cin, val);
```

Nach der Deklaration müssen Sie diese Funktion nur noch global definieren:

```
istream& operator >>(istream& is, myComplex& val ) {
   cout << "Real-Teil       : "; is >> val._real;
   cout << "Imaginärer Teil : "; is >> val._image;
}
```

Das Überladen des Operators `>>` ist aber nicht nur auf die Tastatur beschränkt, sondern lässt sich auch zum Einlesen aus einer Datei »umbiegen«. Hierbei müssen Sie lediglich statt des Streams `cin` einen Datei-Stream wie `ifstream` verwenden und mit einer Datei verbinden:

```
...
// Eingabe-Stream mit Datei verbinden
```

```
ifstream FileCin;
myComplex val;
FileCin.open("MyFile.dat");
// Daten für Objekt aus Datei einlesen
FileCin >> val;
...
```

Dass dies funktioniert, liegt daran, dass `cin` und `ifstream` Instanzen der Basisklasse `istream` sind. Allerdings sei dies nur am Rande erwähnt, da bis jetzt noch nicht die grundlegenden Streams von C++ behandelt wurden. Mehr dazu erfahren Sie ab Abschnitt 7.2, »Ein-/Ausgabe Klassenhierarchie (I/O-Streams)«.

Bevor ich Ihnen das Beispiel zum Einlesen einer Zahl vom Typ `myComplex` demonstriere, soll noch der Ausgabeoperator `<<` überladen werden. Im Grunde können Sie hierbei alles übernehmen, was Sie vom Eingabeoperator `>>` gelesen haben. Nur steht links des Operators `cout` statt `cin`, und es ist ein Objekt der Klasse `ostream` statt `istream`. Sie müssen also statt `istream` für die Eingabe `ostream` für die Ausgabe verwenden. Somit sehen die Deklaration und die Definition der Operatorfunktion wie folgt aus:

```
// Deklaration
friend ostream& operator <<(
           ostream& os, const myComplex& val );
...
// Definition
ostream& operator <<(ostream& os, const myComplex& val ) {
   os << val._real << "+" << val._image << '\n';
   return os;
}
```

Hierzu das komplette Listing:

```
// complex6.cpp
#include <iostream>
using namespace std;

class myComplex {
private:
   double _real;
   double _image;

public:
   myComplex( double val1=0.0, double val2=0.0 ) {
      _real = val1; _image = val2;
   }
```

```
   void print_Complex( ) {
      cout << _real << "+" << _image << '\n';
   }
   friend ostream& operator <<(
                     ostream& os, const myComplex& val );
   friend istream& operator >>(istream& is, myComplex& val);
};

ostream& operator <<(ostream& os, const myComplex& val ) {
   os << val._real << "+" << val._image << '\n';
   return os;
}

istream& operator >>(istream& is, myComplex& val ) {
   cout << "Real-Teil       : "; is >> val._real;
   cout << "Imaginärer Teil : "; is >> val._image;
   return is;
}

int main( void ) {
   myComplex val1, val2, val3;
   cout << "Bitte eine komplexe Zahl eingeben\n";
   cin >> val1;
   cout << val1;
   cout << "Bitte zwei komplexe Zahlen eingeben\n";
   cin >> val2 >> val3;   // Dank Referenz als Rückgabewert
   cout << val2 << val3;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Bitte eine komplexe Zahl eingeben
Real-Teil       : 1.1
Imaginärer Teil : 2.2
1.1+2.2

Bitte zwei komplexe Zahlen eingeben
Real-Teil       : 3.3
Imaginärer Teil : 4.4
Real-Teil       : 5.5
Imaginärer Teil : 6.6
3.3+4.4
5.5+6.6
```

4.5.8 ( )-Operator überladen

Auch der Funktionsoperator `()` lässt sich überladen. Damit können Sie ein Objekt der Klasse als Funktion aufrufen. Daher wird auch von sogenannten Funktionsobjekten gesprochen, ebenso kann man es als bessere Alternative zu den Funktionszeigern bezeichnen. Den Operator `()` für Klassen zu überladen ist in der Tat ein interessantes Feature, das auch in der STL *(Standard Template Library)* intensiv eingesetzt wird. Eine solche Funktion lässt sich praktisch folgendermaßen aufrufen:

```
Objekt(val);
// oder auch
Objekt.operator()(val);
```

Ich möchte den Funktionsoperator bei unserer Klasse `String` zum Einsatz kommen lassen, und zwar folgendermaßen:

```
String string1 = "Adam und Eva";
String string2 = string1(5, 3);
String string3 = string1(9, 3);

String zahl = "100";
long val = 12345, gesamt;
gesamt = val + zahl(10);

// Hexadezimal
cout << zahl(16) << '\n';
// Oktal
cout << zahl(8) << '\n';
```

Mit der Verwendung von

```
String string1 = "Adam und Eva";
String string2 = string1(5, 3);
```

soll an `string2` ein Substring ab Position 5 von `string1` mit 3 Zeichen Länge zugewiesen werden. Natürlich müssen diese Angaben in der Überladung überprüft werden.

Außerdem soll noch eine Funktion implementiert werden, bei der ein Objekt der Klasse `String` in einen `long`-Wert umgewandelt wird. Dabei soll es auch möglich sein, das Zahlensystem anzugeben. Als Basis soll hierbei 16 für hexadezimal, 10 für dezimal und 8 für das Oktalsystem möglich sein:

```
String zahl = "100";
long val = 12345, gesamt;
gesamt = val + zahl(10);
```

Damit wird es möglich, mit der Klasse `String` zu rechnen. Bei dem `()`-Operator haben Sie viele Freiheiten. Entscheidend dafür, was Ihre Funktion machen soll, sind natürlich zunächst der Rückgabewert, der neben einer Klasse auch ein normaler Datentyp sein darf, und die Parameter.

Hierzu das Beispiel des Operators `()` mit der Klasse `String`:

```
// String7.cpp
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class String {
private:
   char *buffer;
   unsigned int len;

public:
   // Konstruktoren
   String( const char* s="") {
      len = strlen(s);
      buffer = new char [len+1];
      strcpy( buffer, s );
   }
   // Destruktor
   ~String() { delete [] buffer; }
   // Kopierkonstruktor - explizit nötig, da
   // dynamischer Speicherbereich verwendet wird
   String( const String& s ) {
      len = s.len;
      buffer = new char [len+1];
      strcpy( buffer, s.buffer );
   }
   // Zugriffsmethode
   char* get_String() const { return buffer; }

   const String operator()(unsigned int pos,
                           unsigned int count) const {
      if( (pos >=0) && (count < len) ) {
         String tmp(*this);
         char *ptr = tmp.buffer+pos;
         // Zeichenkette abschließen
         ptr[count] = '\0';
         return String(ptr);
```

```
      }
      // Bei falscher Angabe
      return String(buffer);
   }

   const long operator()(int base) const {
      // Hexadezimal=16; Dezimal=10; Oktal=8
      if( (base==16) || (base==10) || (base==8) ) {
         unsigned long zahl = strtol(buffer, 0, base);
         return zahl;
      }
      // Im Fehlerfall
      return 0;
    }

   operator char*() const {
      return buffer;
   }
};

int main( void ) {
   String string1 = "Adam und Eva";
   String string2 = string1(5, 3);
   String string3 = string1(9, 3);
   cout << string1 << '\n';
   cout << string2 << '\n';
   cout << string3 << '\n';

   String zahl = "100";
   long val = 12345, gesamt;
   gesamt = val + zahl(10);
   cout << gesamt << '\n';
   // Hexadezimal
   cout << zahl(16) << '\n';
   // Oktal
   cout << zahl(8) << '\n';
   return 0;
}
```

Das Programm bei der Ausführung:

```
Adam und Eva
und
Eva
12445
```

```
256
64
```

Bei der folgenden Methode

```
operator char*() const {
   return buffer;
}
```

handelt es sich nicht um die Überladung des `()`-Operators, sondern um einen Konvertierungsoperator, mit dessen Hilfe es möglich wird, dass ein Aufruf wie

```
cout << string1 << '\n';
```

ohne Überladen des `<<`-Operators möglich wird. Auf solche Typumwandlungen wird noch eingegangen.

[»] **Hinweis**

Bei dem Operator `()` wird niemals eine Referenz zurückgegeben. Der Rückgabewert muss immer ein neues Objekt sein und darf niemals ein Verweis auf ein bestehendes Objekt sein.

4.6 Typumwandlung für Klassen

»Typumwandlung für Klassen« hört sich zunächst recht abenteuerlich an, wenn man sich vorstellt, eine Klasse `Auto` in eine Klasse `Mensch` umzuwandeln. Aber es ist auch möglich, Klassen sowohl explizit als auch implizit umzuwandeln. Wie eine Klasse »umgewandelt« wird, legt allerdings der Programmierer in der Klasse selbst fest. Als Umwandlungen können entweder verschiedene Klassen verwendet werden, oder es werden Umwandlungen zwischen Klassen und Basistypen durchgeführt. Dafür stehen Ihnen zwei Möglichkeiten mit einem Konvertierungskonstruktor und einer Konvertierungsfunktion zur Verfügung.

4.6.1 Konvertierungskonstruktor

Hinter dem Begriff *Konvertierungskonstruktor* verbirgt sich nichts anderes als ein Konstruktor mit Parametern. Somit ist also jeder Konstruktor, den Sie bisher mit Parametern verwendet haben, ein Konvertierungskonstruktor – mit Ausnahme des Kopierkonstruktors.

Der Begriff *Konvertierungskonstruktor* ist anhand der Klasse `String` auch schnell erklärt. Beispielsweise sieht der Konstruktor dieser Klasse folgendermaßen aus:

```
class String {
private:
   char *buffer;
   unsigned int len;

public:
   // Konstruktor
   String( const char* s="") {
      len = strlen(s);
      buffer = new char [len+1];
      strcpy( buffer, s );
   }
};
```

Genau betrachtet, erzeugt dieser Konstruktor aus einem C-String ein Objekt der Klasse `String`, und somit haben wir eine Typumwandlung. Hierzu einige Aufrufmöglichkeiten, die u. a. auch den Konvertierungskonstruktor enthalten:

```
// Standardkonstruktor
String string1;
// Konvertierungskonstruktor char* -> String
String string2("Test-String");
// Kopierkonstruktor
string1 = string2;
// Implizite Konvertierung char* -> String
string1 = "Noch ein Test-String";
// Explizite Konvertierung (Konvertierungskonstruktor)
string1 = String("Test 99");
// Explizite Konvertierung als Cast-Schreibweise
string1 = (String)"More tests";
```

Verwenden Sie den Konvertierungskonstruktor, wird immer ein temporäres Objekt erzeugt, das für die Anweisung verwendet und anschließend wieder zerstört wird.

Sie können also den Konvertierungskonstruktor für explizite und implizite Typumwandlungen benutzen. Bezogen auf die Klasse `myComplex`, bedeutet dies:

```
myComplex val1(1.1, 2.2), val2;
val2 = val1 + (myComplex) 3.3;   // Explizit
val2 = val1 + 3.3 ;              // Implizit
```

Im Beispiel wird davon ausgegangen, dass es noch keinen Operator `+` gibt, der überladen wurde, so dass der implizite Ausdruck `val1 + 3.3` eigentlich einen Fehler ausgeben müsste, weil versucht wird, eine `myComplex`-Zahl mit einer `double`-Zahl zu addieren. Dass dies dennoch funktioniert, liegt daran, dass der Compiler

den Konvertierungskonstruktor heranzieht und versucht, aus der Zahl 3.3 ein temporäres `myComplex`-Objekt zu machen. Dadurch werden `val1` und das temporäre Objekt `myComplex(3.3)` addiert und dann wieder freigegeben.

Selbstverständlich wird neben dem selbstdefinierten Konvertierungskonstruktor gegebenenfalls auch noch die Standardkonvertierung vorgenommen. So ist zum Beispiel auch Folgendes möglich:

```
myComplex val1(1.1, 2.2), val2;
int ival = 3;
// ival wird konvertiert int -> double -> myComplex
val2 = val1 + ival;
```

Hier haben Sie denselben Vorgang wie gehabt, nur dass zunächst der Typ `int` in einen `double`-Wert und dieser vom Konvertierungskonstruktor in ein temporäres `myComplex`-Objekt umgewandelt wird.

4.6.2 Konvertierungsfunktion

Mit einer Konvertierungsfunktion können Sie das Objekt einer Klasse in einen beliebigen Datentyp konvertieren. Dies ist eine Funktion, die festlegt, wie eine Konvertierung erfolgen soll. Die Syntax einer solchen Konvertierungsfunktion lautet:

```
operator zieltyp();
```

Eine solche Konvertierungsfunktion haben Sie bereits bei der Klasse `String` mit

```
operator char*() const {
   return buffer;
}
```

eingesetzt. Durch diese Konvertierung wurde ein Objekt der Klasse `String` zu einem Zeiger auf die Anfangsadresse der Zeichenkette (der Eigenschaft `buffer`) konvertiert. Damit war es praktisch möglich, dass eine Ausgabe mit `cout` wie

```
String string1("Test-String");
cout << string1;
```

ohne Probleme ausgeführt wurde, weil hier `string1` zuvor noch in den entsprechenden Typ umgewandelt wurde (`String -> char*`). Damit ließe sich auch ohne Schwierigkeiten Folgendes verwenden:

```
String string1("Test-String");
char* ptr = string1;
```

Natürlich könnten Sie hierbei auch einen String in ein `int` konvertieren, um zum Beispiel die Länge des Strings zu ermitteln:

```
// String8.cpp
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <cstdio>
using namespace std;

class String {
private:
   char *buffer;
   unsigned int len;

public:
   // Konstruktoren
   String( const char* s="") {
      len = strlen(s);
      buffer = new char [len+1];
      strcpy( buffer, s );
   }
   // Destruktor
   ~String() { delete [] buffer; }
   // Zugriffsmethode
   char* get_String() const { return buffer; }
   // String -> int
   operator unsigned int() const {
      return len;
   }
};

int main( void ) {
   String string1("Test-String");
   int laenge = string1;

   cout << "Anzahl Zeichen in string1: " << laenge << '\n';
   return 0;
}
```

Wollen Sie zum Beispiel bei den komplexen Zahlen Folgendes realisieren

```
myComplex val1(1.1, 2.2);
double val2 = val1;
```

so sieht die Konvertierungsfunktion in der Klasse `myComplex` im `public`-Bereich folgendermaßen aus:

```
operator double() {
   return _real;
}
```

Natürlich stehen Ihnen auch hier wieder alle Tore offen. Haben Sie zum Beispiel eine Schnittstelle für den Programmierer in der Klasse `myComplex` bereitgestellt und dieser versucht, Folgendes auszuführen

```
myComplex val1(1.1, 2.2);
char *val2 = val1;
```

können Sie in der Klasse `myComplex` entweder eine Konvertierungsfunktion wie folgt einbauen

```
operator char*() {
   cout << "myComplex -> char* nicht implementiert\n";
}
```

oder eine entsprechende Funktion implementieren. Natürlich kann der Zieltyp auch aus mehreren Schlüsselwörtern wie zum Beispiel `unsigned int` bestehen.

Bei der Verwendung von Konvertierungsfunktionen müssen Sie Folgendes beachten:

- Eine Konvertierungsfunktion hat keinen Ergebnistyp, da dieser durch den Namen implizit festgelegt wird. Ist beispielsweise der Name der Funktion `operator typ`, so lautet der implizite Name `typ`.
- Die Funktion muss vom aktuellen Objekt ein Objekt vom Zieltyp erzeugen und als Ergebnis zurückgeben.
- Die Konvertierungsfunktion wird immer als Methode ohne Parameter definiert.

4.7 Vererbung (abgeleitete Klassen)

Die Vererbung (oder auch das Ableiten von Klassen) ist ein sehr effizienter und häufig eingesetzter Mechanismus in der C++-Programmierung. Der Vorteil ist, dass hiermit bereits existierende Klassen in neuen Klassen verwendet werden können. Eine so abgeleitete Klasse erbt dann die `public`-Eigenschaften und Methoden der Basisklasse. Die abgeleitete Klasse wird dann gewöhnlich um weitere Eigenschaften und Methoden erweitert.

Eine solche Vererbung von Klassen wird gerne in der Entwicklung von Klassenbibliotheken verwendet, wo gleichartige Methoden und Eigenschaften einer Klasse benötigt werden, sich aber in getrennten Klassen befinden. Selbstverständlich können solche zusammengefassten Klassen noch weiter abgeleitet werden. Aber hierauf wird noch an einer anderen Stelle eingegangen. Der Hauptvorteil für den Programmierer besteht darin, dass dieser den Quellcode für die Klasse nur einmal schreiben und »debuggen« muss. Klug eingesetzt, können Sie solche Klassen immer wieder bei Ihren Projekten verwenden.

Außerdem benötigt der Anwender einer Klasse keinen Quellcode und kann trotzdem mit einer Ableitung der Klasse eine vorhandene Klasse um weitere Eigenschaften und Methoden erweitern. Hierzu ist nur die Schnittstelle zur Basisklasse nötig. Dies wird u. a. in der GUI-Programmierung verwendet.

Ein Objekt vom Typ einer abgeleiteten Klasse ist immer auch ein Objekt vom Typ der Basisklasse. Man sagt auch, dass die abgeleitete Klasse zur Basisklasse in einer *Ist-Beziehung* steht. Ein »Auto« ist ein »Fahrzeug«, und ein »Zug« ist ebenfalls ein »Fahrzeug« (siehe Abbildung 4.3).

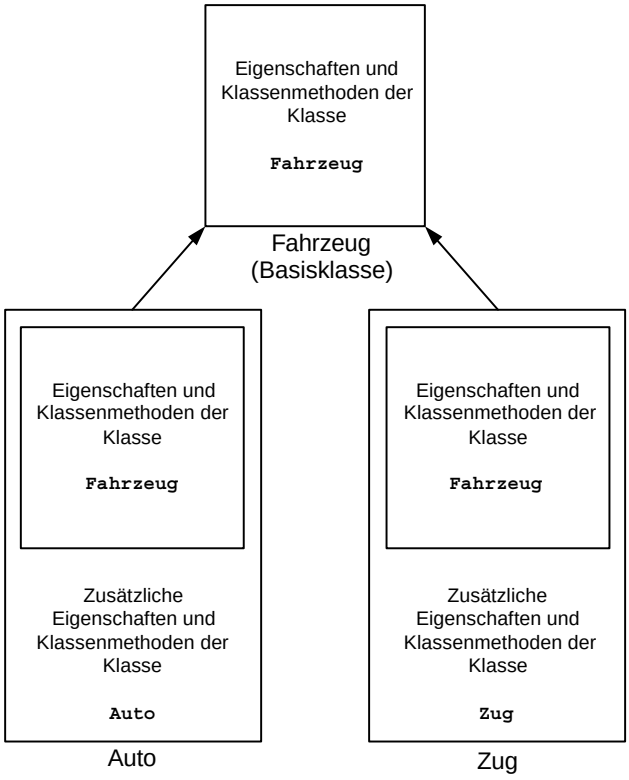

Abbildung 4.3 Ist-Beziehung der abgeleiteten Klasse zur Basisklasse

Das Gegenteil dieser Ist-Beziehung ist die Hat-Beziehung, die entsteht, wenn eine Klasse ein Objekt der anderen Klasse als Element (Eigenschaft) besitzt.

4.7.1 Anwendungsbeispiel (Vorbereitung)

Als Beispiel soll eine kleine Versandfirma dienen, die verschiedene Gegenstände zum Verkauf anbietet. Die Ware selbst hat Eigenschaften wie die Bezeichnung, die Nummer, den Preis und die Anzahl der Waren, die im Lager noch vorhanden sind. Hierzu verwenden wir einfach eine Klasse mit dem Namen `Gegenstand`. Alle Eigenschaften und Methoden werden dabei gleich mit den Deklarationen und Definitionen in eine Datei geschrieben, um die Länge des Codes und der Erklärung nicht zu überstrapazieren. In der Praxis sollte man natürlich wieder die Deklaration von der Definition trennen. Hierzu die Klasse `Gegenstand`, die in einer Header-Datei *gegenstand.h* zusammengefasst und auf das Nötigste beschränkt wurde:

```
// gegenstand.h
#include <iostream>
#include <cstring>
#ifndef _GEGENSTAND_H_
#define _GEGENSTAND_H_
using namespace std;

class Gegenstand {
private:
   char bezeichnung[50];
   unsigned int anzahl;
   unsigned int nummer;
   // Preisangabe in Cent
   unsigned int preis;

public:
   // Konstruktor
   Gegenstand(const char* bezeichnung="",
              unsigned int anzahl=0,
              unsigned int nummer=0, unsigned int preis=0 ){
      strncpy( this->bezeichnung, bezeichnung, 50 );
      this->anzahl = anzahl;
      this->nummer = nummer;
      this->preis  = preis;
   }
   // Destruktor
   ~Gegenstand() { }
   // Zugriffsmethoden
```

```
   // Lesender Zugriff
   const char* get_bezeichnung() const{ return bezeichnung;}
   unsigned int get_anzahl() const { return anzahl; }
   unsigned int get_nummer() const { return nummer; }
   unsigned int get_preis() const { return preis; }
   // Schreibender Zugriff
   void set_bezeichnung( const char* bez ) {
      strncpy( bezeichnung, bez, 50 );
      bezeichnung[50 - 1] = 0;
   }
   void set_anzahl( unsigned int anz ) { anzahl = anz; }
   void set_nummer( unsigned int num ) { nummer = num; }
   void set_preis ( unsigned int pre ) { preis  = pre; }
   // Ausgabe aller Eigenschaften
   void print() const {
      cout << "Artikel   : " << bezeichnung << '\n';
      cout << "Anzahl    : " << anzahl << '\n';
      cout << "Nummer    : " << nummer << '\n';
      cout << "Preis     : " << preis << '\n';
   }
};
#endif
```

Die Klasse `Gegenstand` können Sie jetzt auch ohne größeren Aufwand in einer Hauptfunktion testen:

```
//main.cpp
#include "gegenstand.h"

int main( void ) {
   Gegenstand artikel1;
   artikel1.set_bezeichnung("Deo");
   artikel1.set_anzahl(100);
   artikel1.set_nummer(1);
   artikel1.set_preis(399);
   artikel1.print();
   return 0;
}
```

Mit der Klasse `Gegenstand` erhalten Sie schon einmal die allgemeinen Daten, die eine Ware, die verkauft wird, haben kann. Natürlich lässt sich dies um weitere Eigenschaften erweitern – wenn Gegenstände spezieller werden, ist das im Allgemeinen notwendig.

Verkauft die Versandfirma zum Beispiel auch Bücher, so werden weitere Eigenschaften wie der Titel, der Autor, die Anzahl der Seiten usw. benötigt. Jetzt könn-

ten Sie die Klasse `Gegenstand` um diese Eigenschaften erweitern, oder Sie schreiben einfach eine neue Klasse `Buch`, die dann die Eigenschaften der Klasse `Gegenstand` erhält (erbt) und verwenden kann. Damit bleibt die einfache Wiederverwendbarkeit der Klasse `Gegenstand` für weitere Gegenstände des Versandhauses erhalten, und auch die Abstraktion der Daten ist leicht zu handhaben.

Hierzu also zunächst die Klasse `Buch`, ebenfalls in einer Datei (*buch.h*) und auf das Nötigste beschränkt:

```
// buch.h
#include <iostream>
#include <cstring>
using namespace std;
#ifndef _BUCH_H_
#define _BUCH_H_

class Buch {
private:
   char titel[50];
   char autor[50];
   unsigned int seiten;
   // usw. z.B. Verlag, ISBN etc.

public:
   // Konstruktor
   Buch( const char* titel="",
         const char* autor="", unsigned int seiten=0 ) {
      strncpy( this->titel, titel, 50 );
      this->titel[50 - 1] = 0;
      strncpy( this->autor, autor, 50 );
      this->autor[50 - 1] = 0;
      this->seiten = seiten;
   }
   // Destruktor
   ~Buch() {}
   // Zugriffsmethoden
   // Lesender Zugriff
   const char* get_titel() const { return titel; }
   const char* get_autor() const { return autor; }
   unsigned int get_seiten() const { return seiten; }
   // Schreibender Zugriff
   void set_titel( const char* tit ) {
      strncpy( titel, tit, 50 );
      this->titel[50 - 1] = 0;
   }
   void set_autor( const char* aut ) {
```

```
      strncpy( autor, aut, 50 );
      this->autor[50 - 1] = 0;
   }
   void set_seiten( unsigned int sei ) { seiten = sei; }
   // Ausgabe der Eigenschaften
   void print() {
      cout << "Buchtitel : " << titel << '\n';
      cout << "Autor     : " << autor << '\n';
      cout << "Seiten    : " << seiten << '\n';
   }
};
#endif
```

Auch diese Klasse können Sie ohne weiteres unabhängig von der Klasse `Gegenstand` in der Praxis testen:

```
//main.cpp
#include "buch.h"

int main( void ) {
   Buch buch1;
   buch1.set_titel("C++ von A bis Z");
   buch1.set_autor("Jürgen Wolf");
   buch1.set_seiten(1000);
   buch1.print();
   return 0;
}
```

Bis jetzt haben Sie im Grunde noch nichts anderes als zwei voneinander unabhängige Klassen, die für ein Projekt verwendet werden können. Natürlich wurden diese beiden Klassen in der Absicht erstellt, dass die Basisklasse `Gegenstand` ist und die abgeleitete Klasse `Buch`.

4.7.2 Die Ableitung einer Klasse

Wollen Sie nun die Klasse `Buch` von der (Basis-)Klasse `Gegenstand` ableiten, so ist das sehr einfach. Sie müssen zunächst nur bei der Klasse `Buch` eine Zeile ändern:

```
// buch.h
...
class Buch : public Gegenstand {
   private:
   // ...

   public:
   // ...
};
```

Sie finden hinter der Klasse `Buch` einen Doppelpunkt, gefolgt von der `public`-Vererbung (Zugriffsrechte) und der abschließenden Basisklasse (hier `Gegenstand`).

[»] **Hinweis**

Es soll nicht der Eindruck entstehen, dass die Basisklasse selbst keine abgeleitete Klasse sein kann. Es ist nämlich durchaus möglich, dass die Basisklasse auch eine abgeleitete Klasse ist. In diesem Fall spricht man von einer *indirekten Basisklasse*. Damit lassen sich ganze Klassenhierarchien bilden.

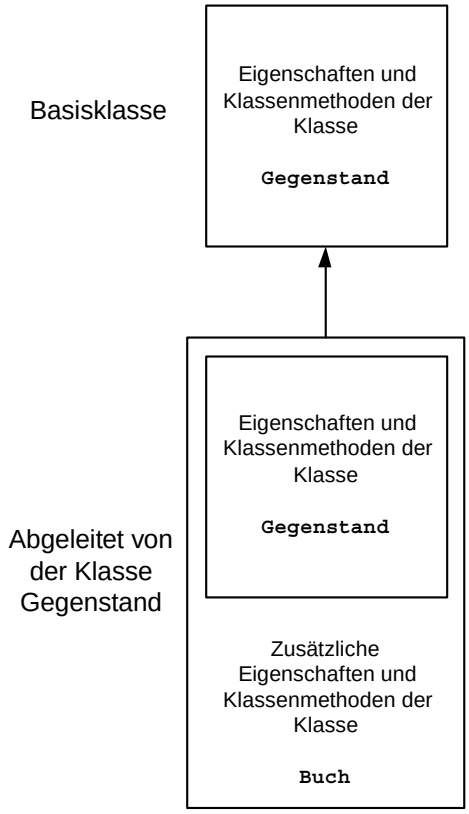

Abbildung 4.4 Basisklasse und abgeleitete Klasse

Die Syntax sieht demnach wie folgt aus:

```
class Klassenname : Zugriffsrechte Basisklasse {
   // ...
}
```

Mit dieser Definition einer abgeleiteten Klasse legen Sie Folgendes fest:

- die Basisklasse, deren Eigenschaften und Methoden vererbt werden
- die Zugriffsrechte auf die Eigenschaften der Basisklasse
- die zusätzlichen Eigenschaften und Methoden, mit denen die Basisklasse erweitert wird

»public«-Zugriffsrechte für eine abgeleitete Klasse

Meistens wird beim Ableiten von der Basisklasse eine `public`-Vererbung definiert, durch die alle öffentlichen `public`-Elemente der Basisklasse auch in der abgeleiteten Klasse zur Verfügung stehen. So können Sie mit Objekten der abgeleiteten Klasse auf Elemente der Basisklasse zugreifen. Im Fall der Klasse `Buch` können Sie alle `public`-Elemente (hier Methoden) der Klasse `Gegenstand` verwenden. Somit können Sie also nicht nur alle `public`-Elemente der Klasse `Gegenstand` über die abgeleitete Klasse `Buch` wie gehabt verwenden, sondern auch noch erweitern.

Hinweis **[«]**

Es ist nach wie vor nicht möglich, dass die abgeleitete Klasse auf die `private`-Elemente der Basisklasse direkt zugreift.

Erbschaft und Erweiterung

Durch die Vererbung der Basisklasse `Gegenstand` an die Klasse `Buch` werden sämtliche Eigenschaften und Methoden vererbt:

```
// buch.h
...
class Buch : public Gegenstand {
   private:
   // ...

   public:
   // ...
};
```

Im Detail erbt die Klasse `Buch` die Bezeichnung, die Nummer, die Anzahl und den Preis der Ware von der Klasse `Gegenstand`. Somit besitzt jedes Objekt vom Typ `Buch` auch die Eigenschaften `bezeichner`, `nummer`, `anzahl` und `preis`. Neu hinzu kommen außerdem bei der Definition der abgeleiteten Klasse `Buch` der Titel, der Autor und die Anzahl der Seiten, also die Eigenschaften `titel`, `autor` und `seiten`. Somit hat ein Objekt vom Typ `Buch` hier sieben Elemente (drei eigene und vier geerbte).

Neben den Eigenschaften der Basisklasse `Gegenstand` werden auch sämtliche Methoden an die Klasse `Buch` weitervererbt. Dies sind neben dem Konstruktor die Zugriffsmethoden `get_bezeichnung()`, `set_bezeichnung()`, `get_anzahl()` usw. In der Klasse `Buch` wiederum wurden weitere neue Methoden definiert, die zu den bereits geerbten Methoden dazukommen.

Besonders interessant erscheint in diesem Zusammenhang, dass beide Klassen eine Methode mit dem Namen `print()` implementiert haben. Somit erbt die Klasse `Buch` von der Klasse `Gegenstand` eine Methode mit gleichem Namen. Man spricht in diesem Fall von einer *Redefinition* – die Methode `print()` wird in der Klasse `Buch` redefiniert. Aber darauf wird später noch näher eingegangen.

Zugriff auf die Elemente (der Basisklasse)

Der Zugriff auf die Eigenschaften innerhalb der abgeleiteten Klasse erfolgt über die Zugriffsmethoden. Dasselbe gilt für den Zugriff auf die Methoden der Basisklasse. Auch hierbei erfolgt der Zugriff auf die privaten Eigenschaften über die Zugriffsmethoden, sofern die Eigenschaften auch im privaten Bereich definiert wurden (was im Grunde immer zu empfehlen ist).

Sobald aber vom Zugriff innerhalb einer Methode die Rede ist, gibt es hierbei Unterschiede. Die Methode `print()` der Klasse `Buch` sieht zum Beispiel wie folgt aus:

```
// Ausgabe der Eigenschaften
void Buch::print() {
   cout << "Buchtitel : " << titel << '\n';
   cout << "Autor     : " << autor << '\n';
   cout << "Seiten    : " << seiten << '\n';
}
```

Um noch die Eigenschaften für die Klasse `Gegenstand` mit auszugeben, können Sie nicht einfach Folgendes einfügen:

```
// Ausgabe der Eigenschaften
void Buch::print() {
   cout << "Buchtitel : " << titel << '\n';
   cout << "Autor     : " << autor << '\n';
   cout << "Seiten    : " << seiten << '\n';
   // !!!!! Fehler !!!!! - Eigenschaften sind private
   cout << "Artikel   : " << bezeichnung << '\n';  // Fehler
   cout << "Anzahl    : " << anzahl << '\n';       // Fehler
   cout << "Nummer    : " << nummer << '\n';       // Fehler
   cout << "Preis     : " << preis << '\n';        // Fehler
}
```

Hier bekommen Sie vom Compiler eine Fehlermeldung zurück, weil versucht wird, die Datenkapselung zu unterlaufen. Die Elemente der Basisklasse `Gegenstand` sind nämlich alle als `private` implementiert. Daher kommen Sie zunächst nicht um die Zugriffsmethoden der Klasse `Gegenstand`, die sich im `public`-Bereich der Klasse befinden, herum:

```
// Ausgabe der Eigenschaften
void Buch::print() {
   cout << "Buchtitel : " << titel << '\n';
   cout << "Autor     : " << autor << '\n';
   cout << "Seiten    : " << seiten << '\n';
   cout << "Artikel   : " << get_bezeichnung() << '\n';
   cout << "Anzahl    : " << get_anzahl() << '\n';
   cout << "Nummer    : " << get_nummer() << '\n';
   cout << "Preis     : " << get_preis() << '\n';
}
```

Sicherlich stellen Sie sich die Frage, woher der Compiler wissen kann, ob die Methode `get_bezeichnung()` oder `get_anzahl()` denn nun eine Methode der eigenen (abgeleiteten) Klasse oder der Basisklasse ist. Der Compiler sucht zunächst in der eigenen (abgeleiteten) Klasse nach einer entsprechenden Methode bzw. nach einem entsprechenden Namen (wird auch als *Name Lookup* bezeichnet). Findet er keinen entsprechenden Namen, sucht der Compiler in der Klassenhierarchie eine Stufe höher, also in der Basisklasse. Handelt es sich um eine indirekte Basisklasse, wird so lange eine Stufe nach oben gegangen, bis eine Methode mit einer entsprechenden Klasse gefunden wurde, oder das Programm lässt sich nicht übersetzen.

In unserem Beispiel wird die Methode `get_bezeichnung()` aus der Basisklasse `Gegenstand` verwendet, weil es in der abgeleiteten Klasse `Buch` keine entsprechende Methode mit diesem Namen gibt. Würde es diesen geben, würde tatsächlich die Methode der Klasse `Buch` verwendet, auch wenn es eine gleichnamige Methode in der Klasse `Gegenstand` gibt.

Hinweis **[«]**

Dass es keine Probleme mit gleichnamigen Methoden gibt, liegt daran, dass die Methoden anhand der Signatur unterschieden werden und nicht anhand des Namens.

4.7.3 Redefinition von Klassenelementen

Redefinition bedeutet, dass in der abgeleiteten Klasse eine Eigenschaft oder eine Methode mit demselben Namen nochmals neu definiert wird, obwohl diese bereits in der Basisklasse vorhanden ist. Wird also eine Eigenschaft oder eine Methode aus der Basisklasse in der abgeleiteten Klasse erneut definiert, überde-

cken die neu definierten Elemente der abgeleiteten Klasse alle gleichnamigen Elemente der Basisklasse. Im Grunde entspricht dies dem Verhalten von globalen und lokalen Variablen, bei denen auch immer die lokalste Variable den Zuschlag erhält. Dies gilt selbstverständlich auch für die Überladung von Methoden; das heißt, Sie können eine Methode der Basisklasse innerhalb der abgeleiteten Klasse auch mehrfach redefinieren (und somit überladen).

Um jetzt aus der abgeleiteten Klasse auf die gleichnamigen Elemente der Basisklasse zuzugreifen, wird der Scope-Operator verwendet. In unserem Beispiel wurde in der Klasse `Buch` die Methode `print()` erneut redefiniert. Würden Sie diese Methode in der abgeleiteten Methode `print()` aufrufen, hätte dies eine Endlosrekursion zur Folge, und das Programm ließe sich ohne Gewalt nicht mehr beenden. Es reicht also aus, die Basisklasse, den Bereichsoperator und die gleichnamige Funktion aufzurufen. Somit sieht unsere neue Version der Methode `print()` in der abgeleiteten Klasse `Buch` folgendermaßen aus:

```
void print() {
   cout << "Buchtitel : " << titel << '\n';
   cout << "Autor     : " << autor << '\n';
   cout << "Seiten    : " << seiten << '\n';
   // Ruft die Methode print der Klasse Gegenstand auf
   Gegenstand::print();
}
```

Natürlich ist die Redefinition von Klassenelementen nicht auf Methoden beschränkt und lässt sich auch auf die Eigenschaften von Klassen anwenden. Allerdings ist dies in der Praxis selten der Fall.

Bisher wurden somit nur in der abgeleiteten Klasse `Buch` die Definition der Ableitung

```
class Buch : public Gegenstand {
   ...
};
```

und die Methode `print()` in der Klasse `Buch` (*buch.h*) verändert. Damit lässt sich jetzt schon ein Objekt vom Typ `Buch` in einem Hauptprogramm verwenden:

```
//main.cpp
#include "gegenstand.h"
#include "buch.h"

int main( void ) {
   Buch buch1;
   Buch* buchPtr;
   Gegenstand gegenstand1("Deo", 100, 1234, 5);
```

```
   buch1.set_titel("C++ von A bis Z");
   buch1.set_autor("Jürgen Wolf");
   buch1.set_seiten(1098);
   buch1.set_bezeichnung("IT-Fachbuch");
   buch1.set_anzahl(2000);
   buch1.set_nummer(32654);
   // Alles ausgeben
   buch1.print();
   cout << "\n\n";
   // Nur den Teil der Klasse Gegenstand ausgeben
   buch1.Gegenstand::print();
   cout << "\n\n";
   // Nur die Methode print der Basisklasse
   // wird hier aufgerufen
   gegenstand1.print();
   // Auch mit einem Zeiger geht es
   cout << "\n\n";
   buchPtr = &buch1;
   buchPtr->print();
   return 0;
}
```

Das Programm bei der Ausführung:

```
Buchtitel : C++ von A bis Z
Autor     : Jürgen Wolf
Seiten    : 1098
Artikel   : IT-Fachbuch
Anzahl    : 2000
Nummer    : 32654
Preis     : 0


Artikel   : IT-Fachbuch
Anzahl    : 2000
Nummer    : 32654
Preis     : 0


Artikel   : Deo
Anzahl    : 100
Nummer    : 1234
Preis     : 5
```

```
Buchtitel : C++ von A bis Z
Autor     : Jürgen Wolf
Seiten    : 1098
Artikel   : IT-Fachbuch
Anzahl    : 2000
Nummer    : 32654
Preis     : 0
```

Im Beispiel können Sie sehen, dass ein Aufruf mit

```
Gegenstand gegenstand1;
...
gegenstand1.print();
```

nur die Methode `print()` der Klasse `Gegenstand` aufruft. Trotz der Redefinition bleibt die Funktionalität der Basisklasse also erhalten. Der Aufruf von

```
Buch buch1;
...
buch1.print();
```

sorgt dafür, dass alle Eigenschaften inklusive der Basisklasse ausgegeben werden. Hierbei wird die redefinierte Methode `print()` der Klasse `Buch` aufgerufen. Will man trotzdem nur den Teil der Basisklasse `Gegenstand` ausgeben, so muss lediglich der Scope-Operator verwendet werden, wobei der Klassenname der Methode vorangestellt wird:

```
// nur die Eigenschaften der Basisklasse ausgeben
buch1.Gegenstand::print();
```

4.7.4 Konstruktoren

Bevor Sie einen speziellen Konstruktor für die abgeleitete Klasse erstellen, müssen Sie zunächst wissen, wie die Daten der abgeleiteten Klasse und der Basisklasse auf- und abgebaut werden.

Wird ein Objekt vom Typ einer abgeleiteten Klasse angelegt, so erfolgt der Aufbau stets von oben nach unten (von der Klassenhierarchie her gesehen) oder auch von innen nach außen. Es wird also immer zuerst der Konstruktor der Basisklasse ausgeführt, gefolgt vom Konstruktor der abgeleiteten Klasse.

Der Abbau eines Objekts geschieht in der genau umgekehrten Reihenfolge. Zuerst wird der Destruktor der abgeleiteten Klasse aufgerufen und anschließend der Destruktor der Basisklasse.

Default-Konstruktor

Den Default-Konstruktor stellt immer die abgeleitete Klasse zur Verfügung, da dieser auch alle Elemente der Basisklasse enthält. Allerdings ist von der Verwendung des Default-Konstruktors bei abgeleiteten Klassen abzuraten.

Bezogen auf die abgeleitete Klasse `Buch`, könnte ein Konstruktor wie folgt aussehen:

```
Buch( const char* bezeichnung="", unsigned int anzahl=0,
      unsigned int nummer=0, unsigned int preis=0,
      const char* titel="", const char* autor="",
      unsigned int seiten=0 ) {
   // Elemente der Basisklasse
   set_bezeichnung(bezeichnung);
   set_anzahl(anzahl);
   set_nummer(nummer);
   set_preis(preis);
   // Elemente der abgeleiteten Klasse
   strncpy( this->titel, titel, 50 );
   this->titel[50 - 1] = 0;
   strncpy( this->autor, autor, 50 );
   this->autor[50 - 1] = 0;
   this->seiten = seiten;
}
```

Hier auch gleich der Grund, warum vom Default-Konstruktor abgeraten wird. In diesem Beispiel wird nämlich erst implizit der Default-Konstruktor für die Basisklasse `Gegenstand` aufgerufen, und anschließend werden diese Eigenschaften mit den `set`-Zugriffsmethoden mit den entsprechenden Werten initialisiert. Also doppelte Arbeit – zunächst wird der Default-Konstruktor der Klasse `Gegenstand` aufgerufen und initialisiert das Teilobjekt mit einem leeren String bzw. mit dem Wert `0` (abhängig von den Eigenschaften). Und dann wird das Teilobjekt nachträglich wieder verändert. Abgesehen vom unnötigen Zeitaufwand setzt das außerdem voraus, dass die Basisklasse einen Default-Konstruktor besitzt. Wenn anschließend die Teilobjekte mit ihren Anfangswerten initialisiert wurden, können erst die Eigenschaften der abgeleiteten Klasse mit Werten versehen werden.

Basisinitialisierer

In unserem Beispiel hat die Basisklasse `Gegenstand` einen eigenen Konstruktor, den man aufrufen kann und sollte, so dass die Elemente der Basisklasse sofort mit den richtigen Werten initialisiert werden. Hierzu verwendet man in C++ den Basisinitialisierer. Der neue Konstruktor der abgeleiteten Klasse `Buch` sieht so aus:

```
// Konstruktor
Buch( const char* titel="", const char* autor="",
```

```
      unsigned int seiten=0 ) {
   strncpy( this->titel, titel, 50 );
   this->titel[50 - 1] = 0;
   strncpy( this->autor, autor, 50 );
   this->autor[50 - 1] = 0;
   this->seiten = seiten;
}

// Konstruktor mit Basisinitialisierer
Buch( const char* bezeichnung, unsigned int anzahl,
      unsigned int nummer, unsigned int preis,
      const char* titel="", const char* autor="",
      unsigned int seiten=0 ) :
      Gegenstand(bezeichnung, anzahl, nummer, preis) {
      strncpy( this->titel, titel, 50 );
      this->titel[50 - 1] = 0;
      strncpy( this->autor, autor, 50 );
      this->autor[50 - 1] = 0;
      this->seiten = seiten;
   }
```

Der Basisinitialisierer wird bei der Definition am Ende der abgeleiteten Klasse durch einen Doppelpunkt getrennt angegeben und besteht aus dem Namen der Klasse und der Argumentenliste mit den Werten, mit denen die Eigenschaften der Basisklasse initialisiert werden sollen.

[»] **Hinweis**

Beachten Sie bitte, dass Sie neben dem Konstruktor mit dem Basisinitialisierer auch noch den herkömmlichen Konstruktor der Klasse `Buch` verwenden und erhalten.

Elementinitialisierer

Natürlich können Sie für die Eigenschaften der abgeleiteten Klasse auch den Elementinitialisierer verwenden, den Sie in Abschnitt 4.4.11, »Teilobjekte initialisieren«, kennengelernt haben – in diesem Fall auch mit dem Basisinitialisierer zusammen, um beide Teilobjekte zu initialisieren. Damit können Sie Basis- und Elementinitialisierer durch Komma getrennt in einer Liste verwenden:

```
// Konstruktor mit Basisinitialisierer
// und Elementinitialisierer
Buch( const char* bezeichnung, unsigned int anzahl,
      unsigned int nummer, unsigned int preis,
      const char* titel="", const char* autor="",
      unsigned int seiten=0 )
      : Gegenstand(bezeichnung, anzahl, nummer, preis) ,
       seiten(seiten) {
```

```
    strncpy( this->titel, titel, 50 );
    this->titel[50 - 1] = 0;
    strncpy( this->autor, autor, 50 );
    this->autor[50 - 1] = 0;
}
```

4.7.5 Destruktoren

Für die abgeleitete Klasse ist nur ein spezieller Destruktor nötig, wenn ein Vorgang rückgängig gemacht werden soll, wie das Freigeben eines dynamisch reservierten Speichers an den Heap. Der Destruktor der Basisklasse hingegen wird immer automatisch ausgeführt und muss nicht explizit aufgerufen werden.

4.7.6 Zugriffsrecht »protected«

Der Zugriff von den abgeleiteten Klassen auf private Elemente der Basisklassen ist weder mit Methoden noch mit `friend`-Funktionen möglich. Die einzige Möglichkeit wäre, ein privates Element der Basisklasse in den `public`-Bereich zu stellen, was aber nicht im Sinne der OOP ist. Somit benötigt man einen Mechanismus, der zwischen `private` und `public` liegt. Eine solche »Verfeinerung« wurde mit dem Schlüsselwort `protected` eingeführt.

Die Zugriffsrechte von `protected` sind von außen nach wie vor dieselben wie bei `private` – mit dem Unterschied, dass jetzt auch Methoden bzw. `friend`-Funktionen einer abgeleiteten Klasse auf die `protected`-Elemente der Basisklasse zugreifen können. Ein Bereich, der mit `protected` deklariert wird, ist für abgeleitete Klassen (und natürlich die eigene Klasse) erreichbar – von außen hingegen lässt sich nach wie vor nichts machen:

Schlüsselwort	Eigene Klasse	Abgeleitete Klasse	Außerhalb
`private`	sichtbar	nicht sichtbar	nicht sichtbar
`protected`	sichtbar	sichtbar	nicht sichtbar
`public`	sichtbar	sichtbar	sichtbar

Tabelle 4.3 Zugriffsschutz und dessen Sichtbarkeit

Auch wenn man mit dem Schlüsselwort `protected` Eigenschaften von der Basisklasse an die abgeleitete Klasse weitervererbt, sollte man Vorsicht walten lassen, weil man hiermit die Datenkapselung etwas lockert. Meistens wird nämlich eine solche `protected`-Deklaration nachträglich eingebaut oder wieder entfernt. Dabei kann es schnell passieren, dass die ein oder andere Methode der abgeleiteten Klasse nicht mehr funktioniert (gerade dann, wenn man das Schlüsselwort `protected` nachträglich entfernt). Hier ein Beispiel dazu:

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

class Basisklasse {
private:
   int int_private;
protected:
   int int_protected;
public:
   int int_public;
   // Konstruktor
   Basisklasse(int val1=0, int val2=0, int val3=0) {
      int_private = val1;
      int_protected = val2;
      int_public = val3;
   }
};

class AbgleiteteKlasse : public Basisklasse {
public:
   // Konstruktor
   AbgeleiteteKlasse (int val1=0, int val2=0, int val3=0) {
      // !!! Fehler, da private !!!
      int_private = val1;
      // Ok, da protected
      int_protected = val2;
      // Ok, da public
      int_public = val3;
   }
   void reset_ints() {
      // !!! Fehler, da private !!!
      int_private = 0;
      // Ok, da protected
      int_protected = 0;
      // Ok, da public
      int_public = 0;
   }
};
```

Vererbung (»public«, »private« und »protected«)

Bisher haben wir nur die `public`-Schnittstelle verwendet, wenn wir bei der abgeleiteten Klasse die Eigenschaften der Basisklasse erben wollten. Allerdings ist der Zugriff auf die Basisklasse nicht auf eine `public`-Vererbung beschränkt. Die `public`-Vererbung sah folgendermaßen aus:

```
class AbgeleiteteKlasse : public Basisklasse
```

Bei einer `public`-Vererbung werden

- die `public`-Elemente der Basisklasse in der abgeleiteten Klasse ebenfalls als `public`-Elemente übernommen.
- die `protected`-Elemente der Basisklasse in der abgeleiteten Klasse ebenfalls als `protected` übernommen und bleiben in der abgeleiteten Klasse auch `protected`.

Eine Stufe darunter kann eine `protected`-Vererbung wie folgt vereinbart werden:

```
class AbgeleiteteKlasse : protected Basisklasse
```

Bei einer solchen `protected`-Vererbung werden

- die `public`-Elemente der Basisklasse in der abgeleiteten Klasse als `protected`-Elemente übernommen.
- die `protected`-Elemente der Basisklasse in der abgeleiteten Klasse ebenfalls als `protected` übernommen und bleiben in der abgeleiteten Klasse auch `protected`.

Und zu guter Letzt ist auch noch eine `private`-Vererbung möglich:

```
class AbgeleiteteKlasse : private Basisklasse
```

Bei einer `private`-Vererbung werden

- die `public`-Elemente der Basisklasse in der abgeleiteten Klasse als `private`-Elemente übernommen.
- die `protected`-Elemente der Basisklasse in der abgeleiteten Klasse als `private`-Elemente übernommen.

Somit stehen bei `private`- und `protected`-Vererbungen in der abgeleiteten Klasse die öffentlichen Schnittstellen der Basisklasse nicht mehr zur Verfügung. Wenn Sie dennoch in die Verlegenheit kommen, einzelne Teile bei der abgeleiteten Klasse in der neuen öffentlichen Schnittstelle übernehmen zu müssen, können Sie dies mit einer Zugriffsdeklaration wie folgt machen:

```
Basisklasse::Element;
```

In der Praxis:

```
class AbgeleiteteKlasse : private Basisklasse {
public:
   Basisklasse::var1;
```

```
protected:
   Basisklasse::var2;
};
```

So steht das `public`-Element `var1` der Basisklasse auch für die abgeleitete Klasse zur Verfügung und ist ebenfalls `public`. Dasselbe kann auch mit einem `protected`-Element (hier im Beispiel mit `var2`) gemacht werden. Auch hierbei ist das `protected`-Element in der abgeleiteten Klasse anschließend `protected`. Die Zugriffsdeklaration beschränkt sich außerdem nicht nur auf die Eigenschaften der Basisklasse, sondern ist auch bei den Methoden gültig.

4.7.7 Typumwandlung abgeleiteter Klassen

Wenn ein Objekt einer abgeleiteten Klasse einer Basisklasse zugewiesen wird, erfolgt implizit eine Typumwandlung in den Typ der Basisklasse. So können Sie Objekte vom Typ `Buch` Objekten vom Typ `Gegenstand` zuweisen (siehe Abbildung 4.5).

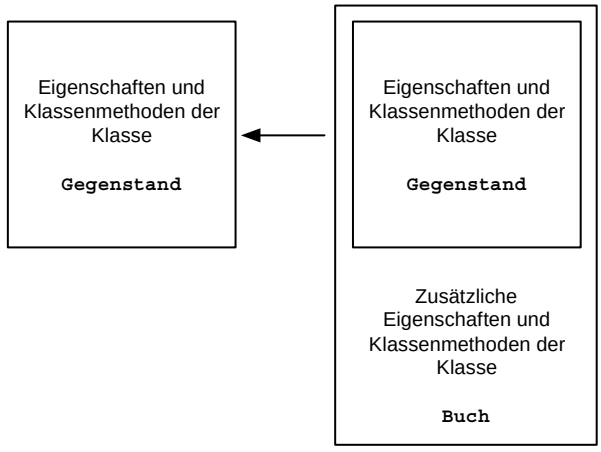

Abbildung 4.5 Zuweisung der Klasse »Buch« an die Basisklasse »Gegenstand«

Wird ein Objekt einer abgeleiteten Klasse einem Objekt der Basisklasse zugewiesen, werden (logischerweise) nur die Eigenschaften der Basisklasse zugewiesen. Die anderen Komponenten werden nicht berücksichtigt:

```
//main1.cpp
#include "gegenstand.h"
#include "buch.h"

int main( void ) {
   Gegenstand gegenstand1;
```

```
   Buch buch1( "IT-Fachbuch", 100, 123, 0,
               "C++ von A bis Z", "J. Wolf", 1000 );
   // Implizite Typumwandlung
   gegenstand1 = buch1;
   gegenstand1.print();
   return 0;
}
```

Das Programm bei der Ausführung:

```
Artikel   : IT-Fachbuch
Anzahl    : 100
Nummer    : 123
Preis     : 0
```

Eine umgekehrte Zuweisung von einem Objekt der Basisklasse an ein Objekt der abgeleiteten Klasse wie beispielsweise

```
// AbgeleiteteKlasse = Basisklasse - Fehler
buch1 = gegenstand1;
```

ist nicht möglich, weil die Eigenschaften der abgeleiteten Klasse nicht zugeordnet werden können und undefiniert wären. Natürlich ist eine Zuweisung trotzdem möglich. Sie könnten zum Beispiel in der abgeleiteten Klasse den Zuweisungsoperator entsprechend überladen. Es würde aber auch genügen, wenn ein Konstruktor der Basisklasse als Parameter eine Referenz auf ein Basisobjekt hat. Die Typumwandlung wird dann vom Konstruktor übernommen.

Diese Ist-Beziehung zwischen der abgeleiteten Klasse und der Basisklasse gilt selbstverständlich auch für die Verwendung von Zeigern (bzw. Basisklassenzeigern) und Referenzen. So können Sie auch Basisklassenzeiger einsetzen, die auf Objekte abgeleiteter Klassen verweisen. Wobei hier auch nur die öffentlichen Schnittstellen der Basisklasse verwendet werden können. Es können also mit einem Basisklassenzeiger keine Methoden aufgerufen werden, die in der abgeleiteten Klasse (re)definiert wurden. Ein Zeiger kann letztendlich auch nur auf etwas zeigen, dessen Typ er selbst repräsentiert. Dasselbe gilt analog beim Arbeiten mit Referenzen. Hierzu ein Beispiel:

```
//main2.cpp
#include "gegenstand.h"
#include "buch.h"

int main( void ) {
   // Basisklassenzeiger
   Gegenstand* gegenstandPtr;
   Buch buch1( "IT-Fachbuch", 100, 123, 0,
               "C++ von A bis Z", "J. Wolf", 1000 );
```

```
   // Adresse zuweisen
   gegenstandPtr =& buch1;
   gegenstandPtr->print();
   cout << '\n';
   // Referenz auf Basisobjekte
   Gegenstand& gegenstandRef = buch1;
   gegenstandRef.print();
   return 0;
}
```

Das Programm bei der Ausführung:

```
Artikel   : IT-Fachbuch
Anzahl    : 100
Nummer    : 123
Preis     : 0

Artikel   : IT-Fachbuch
Anzahl    : 100
Nummer    : 123
Preis     : 0
```

Explizite Typumwandlung

Wollen Sie dennoch, dass alle Elemente bei der Zuweisung einer abgeleiteten Klasse an eine Basisklasse ausgegeben werden, können Sie dies mit einer expliziten Typumwandlung wie folgt erzwingen:

```
// Basisklassenzeiger
Gegenstand* gegenstandPtr;
Buch buch1( "IT-Fachbuch", 100, 123, 0,
            "C++ von A bis Z", "J. Wolf", 1000 );
// Adresse zuweisen
gegenstandPtr =& buch1;
// Explizite Typumwandlung
(static_cast<Buch*>(gegenstandPtr))->print();
```

Die Ausgabe:

```
Buchtitel : C++ von A bis Z
Autor     : J. Wolf
Seiten    : 1000
Artikel   : IT-Fachbuch
Anzahl    : 100
Nummer    : 123
Preis     : 0
```

Beachten Sie aber, dass eine explizite Typumwandlung auch ihre Tücken haben kann. Denn falls hier `gegenstandPtr` nicht auf ein Objekt vom Typ `Buch` zeigt, wird mit der Methode `print` der Klasse `Gegenstand` auf einen undefinierten Speicherbereich zugegriffen, der nicht zum Objekt gehört.

4.7.8 Klassenbibliotheken erweitern

Es wurde bereits erwähnt, dass der Vorteil beim Ableiten einer Klasse neben der vereinfachten Datenabstraktion darin besteht, eine bereits vorhandene Klassenbibliothek, deren Quellcode Sie nicht benötigen, erweitern zu können – was theoretisch auch die Standardbibliotheken mit einschließt.

Wenn Sie eine Klassenbibliothek erweitern wollen, benötigen Sie den übersetzten Quellcode, der entweder in Form einer Objektdatei (*.obj/.o*) oder einer Bibliotheksdatei (*.lib/.a*) vorliegt, sowie die entsprechende Header-Datei (*.h*). Im Grunde haben Sie ja von der Standardbibliothek auch nicht mehr.

Jetzt müssen Sie nur noch die Header-Datei in Ihren Quellcode einbinden und dem Linker die Objektdatei (*.obj/.o*) oder Bibliotheksdatei (*.lib/.a*) mitteilen:

```
// meinProjekt.cpp
#include "original_klasse.h"
...
int main ( void ) {
   // ...
}
```

Hier haben Sie die Header-Datei *original_klasse.h* mit eingebunden. Im Beispiel soll außerdem noch eine Objektdatei mit dem Namen *original_klasse.obj* vorhanden sein. Diese linken Sie nun ebenfalls zum Programm hinzu.

Sie wollen jetzt also die Originalklasse erweitern. Hierzu erstellen Sie zunächst eine eigene neue Header-Datei für eine neue Klasse (im Beispiel soll der Name »meine_Klasse.h« verwendet werden). Hierbei binden Sie die neue Header-Datei *original_klasse.h* ein und deklarieren eine Ableitung der Originalklasse:

```
// meine_Klasse.h
#ifndef MEINE_KLASSE_H
#define MEINE_KLASSE_H
#include "original_klasse.h"
class meineKlasse : public originalKlasse {
   // Deklarationen
};
#endif
```

Jetzt erstellen Sie gewöhnlich noch eine weitere Quelldatei (*.cpp*), in der Sie die Definitionen der Methoden der neuen Klasse vornehmen.

```
// meine_Klasse.cpp
#include "meine_klasse.h"
...
// Definitionen der Methoden
```

Jetzt müssen Sie in Ihrem eigenen Programm die neue Header-Datei *meine_Klasse.h* mit einbinden und selbstverständlich auch die Quelldatei *meine_Klasse.cpp* zum Projekt hinzufügen:

```
// meinProjekt.cpp
#include "meine_klasse.h"
...
int main ( void ) {
   // ...
}
```

4.8 Polymorphismus

Polymorphie – ein auf den ersten Blick seltsames Wort für die Informatik (Polymorphie = Vielgestaltigkeit), das sich aber schnell erklären lässt. In einem der vorangegangenen Abschnitte (Abschnitt 4.7.7, »Typumwandlung abgeleiteter Klassen«, Beispiel *main2.cpp*) wurde folgender Code verwendet:

```
// Basisklassenzeiger
Gegenstand* gegenstandPtr;
Buch buch1( "IT-Fachbuch", 100, 123, 0,
            "C++ von A bis Z", "J. Wolf", 1000 );
// Adresse zuweisen
gegenstandPtr =& buch1;
gegenstandPtr->print();
```

In diesem Beispiel haben Sie die Basisklasse `Gegenstand` und die abgeleitete Klasse `Buch` verwendet, wobei beide Klassen die Methode `print()` enthalten. Und trotzdem wird in diesem Beispiel die print-Methode der Klasse `Gegenstand` aufgerufen. Zur Übersetzungszeit bindet der Compiler dabei die Methode `print()` an die Klasse `Gegenstand`, weil die Variable `gegenstandPtr` vom Typ `Gegenstand` ist. Diese Art des Bindens nennt man eine *statische Bindung*, sie ist unveränderbar.

In der objektorientierten Programmiersprache ist es nun möglich, dass die Variable `gegenstandPtr` auch eine andere »Gestalt« annimmt, indem diese auf

Objekte der abgeleiteten Klasse zeigt und somit auch ein `Buch`-Objekt oder ein beliebiges anderes Objekt darstellt.

Natürlich kann eine solche »Gestalt« nicht mehr statisch erfolgen, sondern sie wird dynamisch zur Laufzeit und nicht mehr zur Übersetzungszeit gebunden. Hierbei wird zunächst beim Aufruf einer solchen Methode vom System der Typ des Objekts untersucht, auf das die Methode angewandt werden soll. Abhängig vom Typ wird diese Methode mit `Gegenstand::print()` oder `Buch::print()` ausgewählt. Hierbei handelt es sich um eine *dynamische Bindung*. Solche dynamischen Bindungen werden mit virtuellen Methoden realisiert.

4.8.1 Statische bzw. dynamische Bindung

Beide Begriffe sind nicht schwer zu verstehen, aber es stellt sich die Frage, wie der Compiler das macht? Beim Aufruf gewöhnlicher Methoden steht zum Zeitpunkt der Übersetzung die Adresse der Methode fest, die aufgerufen werden soll. Die Adresse wird hierbei fest im Maschinencode des Methodenaufrufs gespeichert. Bei einer statischen Bindung spricht man daher auch von einer *frühen Bindung*.

Virtuelle Methoden werden über Zeiger bzw. Referenzen realisiert und wissen zur Laufzeit noch nicht, welche Methode ausgeführt werden soll. Hierbei können verschiedene virtuelle Methoden aufgerufen werden. Welche Methode das ist, hängt immer davon ab, welches Objekt der Zeiger bzw. die Referenz adressiert. Der Compiler erzeugt einen Code, der erst zur Laufzeit gebunden wird. Diese dynamische Bindung bezeichnet man auch als *späte Bindung*.

Eine solche dynamische Bindung lässt sich hervorragend verwenden, wenn man eine Klassenbibliothek erweitern will, deren Quellcode man nicht besitzt; man kann sogar nachträglich einen bereits übersetzten Code erweitern. Dies wird gewöhnlich bei der Erweiterung kommerzieller Bibliotheken eingesetzt, von denen man meistens nur die Header-Dateien und die übersetzten Module (Objektdateien; *.obj*) besitzt. Von der kommerziellen Klasse muss anschließend nur noch die eigene Klasse abgeleitet werden, und neue virtuelle Methoden müssen dafür definiert werden. Dank der dynamischen Bindung lassen sich so von den Methoden der Klassenbibliothek neue Methoden aufrufen.

4.8.2 Virtuelle Methoden

Sie haben gelernt, dass ein `Buch`-Objekt ein `Gegenstand`-Objekt ist. Bisher haben Sie erfahren, dass ein `Buch`-Objekt die Eigenschaften und Methoden seiner Basisklasse (`Gegenstand`) geerbt hat. Diese Ist-Beziehung ist aber noch nicht am Ende, wie Sie in Abschnitt 4.7.7, »Typumwandlung abgeleiteter Klassen«, erfahren

haben, wo eine abgeleitete Klasse dem Zeiger einer Basisklasse zugewiesen wurde. Beispielsweise ist auch Folgendes möglich:

```
Gegenstand* gegenstand1 = new Buch;
```

Damit erzeugen Sie ein neues `Buch`-Objekt auf dem Heap. Ungewöhnlich erscheint auch hier, dass der Zeiger vom Typ `Gegenstand` ist. Aber das ist durchaus sinnvoll, da ein Buch ja auch ein Gegenstand ist.

Über diesen Zeiger können Sie jetzt jede Methode der Klasse `Gegenstand` aufrufen, wie Sie dies ja bereits in folgendem Codeausschnitt gesehen haben:

```
// Basisklassenzeiger
Gegenstand* gegenstandPtr;
Buch buch1( "IT-Fachbuch", 100, 123, 0,
            "C++ von A bis Z", "J. Wolf", 1000 );
// Adresse zuweisen
gegenstandPtr =& buch1;
gegenstandPtr->print();
```

Der Nachteil war, dass so nur die Eigenschaften des Objekts vom Typ `Gegenstand` ausgegeben wurden. Es wurde bereits erwähnt, dass man hierbei theoretisch die Basisklasse um eine Eigenschaft (eine Referenz) erweitern könnte, die sich den Typ des Objekts merkt. Zusätzlich käme noch eine `switch`-Abfrage hinzu, die dann die entsprechende Methode aufrufen würde. Allerdings hätten Sie zum einen das Problem, dass Sie bei jeder weiteren Ableitung den Code um eine `case`-Marke erweitern und somit das Programm auch neu übersetzen müssten. Zum anderen würde dies eine Menge weiterer Codezeilen und Überprüfungen bedeuten, was auch zusätzliche Rechenzeit für den Rechner mit sich bringt. Sie werden sich vielleicht gefragt haben, warum ich hierzu kein Beispiel erstellt und demonstriert habe. Das war nicht nötig, weil es hierfür die virtuellen Methoden gibt.

Die Lösung für die Polymorphie (Vielgestaltigkeit) lautet also virtuelle Methoden. Damit ist es praktisch möglich, dass eine in `Buch` redefinierte Methode korrekt aufgerufen wird. Wenn ein Basisklassenzeiger `Gegenstand` auf ein Objekt der abgeleiteten Klasse `Buch` zeigt, soll es möglich sein, dass mit der Anweisung

```
gegenstandPtr =& buch1;
gegenstandPtr->print();
```

alle Eigenschaften des Objekts (also die Eigenschaften von `Buch`) ausgegeben werden. Auf der anderen Seite sollen bei einem Aufruf wie

```
gegenstandPtr =& einGegenstand;
gegenstandPtr->print();
```

nur die Eigenschaften des Objekts vom Typ `Gegenstand` ausgegeben werden.

Hierzu müssen Sie nur die entsprechende(n) Methode(n) mit dem Schlüsselwort `virtual` deklarieren. Die Definition unterscheidet sich nicht von der Definition der anderen Methoden.

Hinweis [«]

Konstruktoren können nicht als `virtual` deklariert werden.

Hier nochmals die Klasse `Gegenstand` (gekürzt) mit der virtuellen Methode `print()`:

```
// gegenstand.h
#include <iostream>
#include <cstring>
#ifndef _GEGENSTAND_H_
#define _GEGENSTAND_H_
using namespace std;

class Gegenstand {
private:
   ...

public:
   ...
   // Virtuelle Methode
   virtual void print() const {
      cout << "Artikel   : " << bezeichnung << '\n';
      cout << "Anzahl    : " << anzahl << '\n';
      cout << "Nummer    : " << nummer << '\n';
      cout << "Preis     : " << preis << '\n';
   }
};
#endif
```

Jetzt noch die Klasse `Buch`, die ebenfalls eine redefinierte virtuelle Methode `print()` enthält:

```
// buch.h
#include <iostream>
#include <cstring>
#include "gegenstand.h"
#ifndef _BUCH_H_
#define _BUCH_H_

class Buch : public Gegenstand {
private:
   ...
```

```
public:
   ...
   // Implizit virtuell
   virtual void print() const {
      cout << "Buchtitel : " << titel << '\n';
      cout << "Autor     : " << autor << '\n';
      cout << "Seiten    : " << seiten << '\n';
      Gegenstand::print();
   }
};
#endif
```

Jetzt haben Sie zwei Funktionen als virtuell deklariert. Diese Technik wird gewöhnlich dann verwendet, wenn eine abgeleitete Klasse eine eigene gleichnamige Methode zur Basisklasse definiert. Damit erreichen Sie, dass der Compiler jetzt veranlasst wird, immer die richtige Methode zum zugehörigen Objekt aufzurufen.

Damit das funktioniert, ist es wichtig, dass das Objekt über einen Zeiger oder eine Referenz auf die Basisklasse angesprochen wird. Das ist der Unterschied zu den herkömmlichen Methoden. Da Sie nicht erwarten können, dass der Programmierer immer zuerst einen Basisklassenzeiger auf ein Objekt wie folgt verwendet

```
gegenstandPtr =& buch1;
gegenstandPtr->print();
```

können Sie alternativ auch eine globale Funktion schreiben. Damit stellen Sie sicher, dass die virtuelle Methode immer mit einem Basisklassenzeiger aufgerufen wird:

```
void myPrint( Gegenstand* g ) {
   g->print();
}
```

Diese Funktion können Sie nun wie folgt aufrufen:

```
myPrint( &buch1 );
```

Natürlich können Sie auch diese Funktion mit einer Referenz als Parameter implementieren:

```
void myPrint( Gegenstand& g ) {
   g.print();
}
```

Dabei benötigt man beim Aufruf der Funktion natürlich keinen Adressoperator mehr:

```
myPrint( buch1 );
```

Hierzu die komplette Hauptfunktion mit einigen Beispielen, wie die virtuellen Methoden aufgerufen werden können:

```
// main.cpp
#include "gegenstand.h"
#include "buch.h"

void myPrint( Gegenstand* g ) {
   g->print();
}

int main( void ) {
   // Basisklassenzeiger
   Gegenstand* gegenstandPtr;
   Gegenstand gegenstand1( "Roman", 100, 124, 0 );
   Buch buch1( "IT-Fachbuch", 100, 123, 0,
               "C++ von A bis Z", "J. Wolf", 1000 );

   gegenstandPtr =& buch1;
   gegenstandPtr->print();
   cout << '\n';

   gegenstandPtr =& gegenstand1;
   gegenstandPtr->print();
   cout << '\n';

   myPrint( &buch1 );
   cout << '\n';

   myPrint( &gegenstand1 );
   return 0;
}
```

Das Programm bei der Ausführung:

```
Buchtitel : C++ von A bis Z
Autor     : J. Wolf
Seiten    : 1000
Artikel   : IT-Fachbuch
Anzahl    : 100
Nummer    : 123
```

```
Preis     : 0

Artikel   : Roman
Anzahl    : 100
Nummer    : 124
Preis     : 0

Buchtitel : C++ von A bis Z
Autor     : J. Wolf
Seiten    : 1000
Artikel   : IT-Fachbuch
Anzahl    : 100
Nummer    : 123
Preis     : 0
Artikel   : Roman
Anzahl    : 100
Nummer    : 124
Preis     : 0
```

An der Ausgabe des Programms können Sie feststellen, dass immer die richtige virtuelle Methode aufgerufen wird.

4.8.3 Virtuelle Methoden redefinieren

Die Verwendung der virtuellen Methoden kann auf den ersten Blick ziemlich komplex sein. Eine virtuelle Methode der Basisklasse muss zunächst nicht in der abgeleiteten Klasse als virtuell redefiniert werden. Die abgeleitete Klasse erbt sowieso die virtuelle(n) Methode(n). Ein einfaches Beispiel:

```
// virtual1.cpp
#include <iostream>
using namespace std;

class Basisklasse {
public:
   virtual void funktion1() const {
      cout << "Basisklasse::funktion1()\n";
   }
};

class abgeleiteteKlasse : public Basisklasse {
public:
   void funktion1() const {
      cout << "Abgeleitet::funktion1\n";
```

```
   }
};

int main( void ) {
   abgeleiteteKlasse abgelObjekt;
   Basisklasse* basisPtr;
   basisPtr =& abgelObjekt;
   // Zugriff über den Basisklassenzeiger
   basisPtr->funktion1();
   // Zugriff über das Objekt der Klasse abgeleiteteKlasse
   abgelObjekt.funktion1();
   return 0;
}
```

Die Ausgabe des Programms:

```
Abgeleitet::funktion1
Abgeleitet::funktion1
```

Ohne das Schlüsselwort `virtual` vor der Methode `funktion1()` in der Basisklasse sähe die Ausgabe folgendermaßen aus:

```
Basisklasse::funktion1()
Abgeleitet::funktion1
```

Auf diese Weise kann man festlegen, dass »einmal virtuell immer virtuell« bedeutet. Eine neue redefinierte Version ist auch wieder virtuell. Zwar kann bei der redefinierten Version auch wieder das Schlüsselwort `virtual` verwendet werden, muss aber nicht.

Andersherum hat aber Folgendes keinen Effekt:

```
class Basisklasse {
   public:
   void funktion1() const {
      cout << "Basisklasse::funktion1()\n";
   }
};

class abgeleiteteKlasse : public Basisklasse {
   public:
   virtual void funktion1() const {
      cout << "Abgeleitet::funktion1\n";
   }
};
```

Nur weil Sie hier die Methode der abgeleiteten Klasse als virtuell deklariert haben, bedeutet dies nicht, dass die Basisklasse automatisch virtuell ist.

[»] **Hinweis**

Bitte beachten Sie, dass eine Redefinition kein Polymorphismus ist. Erst durch das Schlüsselwort `virtual` zeigt man an, dass die Entscheidung über die aufgerufene Methode zur Laufzeit auf Basis des aktuellen Objekts erfolgt. Wenn Sie das Schlüsselwort `virtual` weglassen, wird aus Polymorphismus wieder eine einfache Redefinition.

[»] **Hinweis**

Außerdem sollte man nicht den Fehler machen, Redefinition und Überladung in einen Topf zu werfen, auch wenn beide Verfahren Ähnlichkeiten aufweisen. Bei einer Überladung erzeugen Sie mehrere Methoden mit dem gleichen Namen, aber mit einer unterschiedlichen Signatur. Bei einer Redefinition erzeugen Sie in der abgeleiteten Klasse eine Methode mit gleichem Namen wie die Methode in der Basisklasse und mit der gleichen Signatur.

Signatur

Dennoch wird nicht einfach vererbt, wenn etwas nicht zusammenpasst. Es genügt schließlich nicht, eine Methode in der Basisklasse als virtuell zu kennzeichnen und eine weitere Methode in der abgeleiteten Klasse mit demselben Namen zu deklarieren.

Die redefinierte Methode in der abgeleiteten Klasse benötigt neben dem gleichen Namen auch dieselbe Signatur (Parameter) und auch den gleichen Rückgabewert wie die gleichnamige Methode der Basisklasse. Besitzt diese redefinierte virtuelle Methode eine andere Signatur, so wird lediglich eine weitere Methode mit gleichem Namen erzeugt, die allerdings nicht virtuell ist – es ist also kein Fehler, wenn eine Methode virtuell ist und in der abgeleiteten Klasse mit einer anderen Signatur redefiniert ist. Dazu ein Beispiel, das den hier beschriebenen Vorgang nochmals demonstriert:

```
// virtual2.cpp
#include <iostream>
using namespace std;

class Basisklasse {
public:
   void funktion1() const {
      cout << "Basisklasse::funktion1()\n";
   }
   virtual void funktion2() const {
      cout << "Basisklasse::funktion2()\n";
   }

   virtual void funktion3() const {
```

```
      cout << "Basisklasse::funktion3()\n";
   }
};

class abgeleiteteKlasse : public Basisklasse {
public:
   void funktion1() const {
      cout << "Abgeleitet::funktion1\n";
   }
   // Virtuell
   void funktion2() const {
      cout << "Abgeleitet::funktion2\n";
   }
   // Nicht virtuell, da andere Signatur, weil Parameter
   void funktion3( double dwert=0.0 ) const {
      cout <<"(" << dwert << ") Abgeleitet::funktion2\n";
   }
};

int main( void ) {
   abgeleiteteKlasse abgelObjekt;
   Basisklasse* basisPtr;
   basisPtr =& abgelObjekt;

   // Zugriff über den Basisklassenzeiger
   basisPtr->funktion1();  // Basisklasse
   basisPtr->funktion2();  // Abgeleitete Klasse
   basisPtr->funktion3();  // Basisklasse

   // Zugriff über das Objekt der Klasse abgeleiteteKlasse
   abgelObjekt.funktion1();
   abgelObjekt.funktion2();
   abgelObjekt.funktion3();

   // Zugriff über abgleitetes Objekt auf die Basisklasse
   abgelObjekt.Basisklasse::funktion3();
   return 0;
}
```

Das Programm bei der Ausführung:

```
Basisklasse::funktion1()
Abgeleitet::funktion2
Basisklasse::funktion3()
Abgeleitet::funktion1
```

```
Abgeleitet::funktion2
(0) Abgeleitet::funktion2
Basisklasse::funktion3()
```

Rückgabewert

Da sich erst zur Laufzeit entscheidet, welche virtuelle Methode ausgeführt wird, muss auch der Rückgabewert der verschiedenen virtuellen Methoden gleich sein. Eine Ausnahme gibt es, und zwar wenn eine virtuelle Funktion einen Zeiger bzw. eine Referenz auf die Basisklasse selbst zurückliefert – dann darf auch die neu definierte virtuelle Methode einen Zeiger bzw. eine Referenz auf eine von der Basisklasse abgeleitete Klasse zurückgeben. Verwenden Sie dann eine virtuelle Funktion über den Basisklassenzeiger oder über eine Referenz auf die Basisklasse, wird der Rückgabewert implizit in den Typ umgewandelt, den die Methode der Basisklasse hat. Das hört sich schlimmer an, als es ist, daher hierzu wieder ein Beispiel:

```
// virtual3.cpp
#include <iostream>
using namespace std;

class Basisklasse {
public:
   virtual Basisklasse& funktion1() {
      cout << "Basisklasse::funktion1()\n";
      return *this;
   }
};

class abgeleiteteKlasse : public Basisklasse {
public:
   // virtuell
   abgeleiteteKlasse& funktion1() {
      cout << "abgeleiteteKlasse::funktion1()\n";
      return *this;
   }
};

int main( void ) {
   abgeleiteteKlasse abgelObjekt;
   Basisklasse* basisPtr;
   basisPtr =& abgelObjekt;

   // Zeiger
   basisPtr->funktion1();  // abgeleiteteKlasse
```

```
   // Referenz
   Basisklasse& basisRef = abgelObjekt;
   basisRef.funktion1();   // abgeleiteteKlasse

   // normales Objekt
   Basisklasse basisVar = abgelObjekt;
   basisVar.funktion1();   // Basisklasse
   return 0;
}
```

Das Programm bei der Ausführung:

```
abgeleiteteKlasse::funktion1()
abgeleiteteKlasse::funktion1()
Basisklasse::funktion1()
```

Zugriffsrechte

Natürlich kann man die virtuellen Methoden, wie andere Methoden auch, in `public`-, `protected`- und `private`-Bereiche aufteilen. Es gibt jedoch eine Besonderheit der virtuellen Methoden. Deklarieren Sie eine virtuelle Funktion in der abgeleiteten Klasse als `private`, so ist diese Methode (wie sonst auch üblich) nicht über ein Objekt des entsprechenden Typs ansprechbar. Über den Zugriff eines Basisklassenzeigers oder einer Referenz auf die Basisklasse können Sie dennoch auf die private virtuelle Methode der Basisklasse zugreifen. Damit erzwingen Sie den Zugriff auf die private virtuelle Methode der abgeleiteten Klasse über einen Zeiger oder eine Referenz auf die Basisklasse. Hierbei stellt die Basisklasse die »offene« Schnittstelle für die private Methode der abgeleiteten Klasse dar. Das Beispiel hierzu:

```
// virtual4.cpp
#include <iostream>
using namespace std;

class Basisklasse {
public:
   virtual void funktion1() const {
      cout << "Basisklasse::funktion1()\n";
   }
};

class abgeleiteteKlasse : public Basisklasse {
private:
   // virtuell
  void funktion1() const {
```

```
      cout << "abgeleiteteKlasse::funktion1()\n";
   }
};

int main( void ) {
   abgeleiteteKlasse abgelObjekt;
   Basisklasse* basisPtr;
   basisPtr =& abgelObjekt;

   // Zeiger
   basisPtr->funktion1();

   // Referenz
   Basisklasse& basisRef = abgelObjekt;
   basisRef.funktion1();

   // !!! Nicht möglich, da private !!!
   // abgelObjekt.funktion1();
   return 0;
}
```

Das Programm bei der Ausführung:

```
abgeleiteteKlasse::funktion1()
abgeleiteteKlasse::funktion1()
```

4.8.4 Arbeitsweise virtueller Methoden

Bisher haben Sie die virtuellen Methoden einfach eingesetzt. Man sollte aber auch ein wenig hinter die Kulissen virtueller Methoden schauen, um zu sehen, was bei einer solchen dynamischen Bindung intern abläuft.

Wer viel mit C programmiert hat, wird schon erahnen, dass hierbei Funktionszeiger am Werke sind. Hierzu legt der Compiler für jede Klasse, die mindestens eine virtuelle Methode besitzt, eine virtuelle Methodentabelle an (oder kurz VMT für *Virtual Method Table*). Diese Tabelle ist im Grunde nur ein Array von Funktionszeigern, die die virtuellen Methoden einer entsprechenden Klasse adressieren. Jede virtuelle Methode erhält in dieser Tabelle einen Eintrag mit ihrer Adresse. Jede dieser virtuellen Methoden in den Tabellen von Basis- und abgeleiteten Klassen besitzt dabei immer den gleichen Index.

Um auf diese virtuelle Methodentabelle zuzugreifen, wird außerdem noch ein Zeiger benötigt, ein sogenannter *virtueller Methodenzeiger* (*VMT-Zeiger*). Ein solcher Zeiger wird ebenfalls intern vom Compiler erzeugt und für jedes Objekt zur

Verfügung gestellt. Mit diesem versteckten Zeiger wird über die Verwendung eines Objekts auf die virtuelle Methode der entsprechenden Klasse zugegriffen.

Für die dynamische Bindung wird zunächst im referenzierten Objekt der virtuelle Zeiger auf die virtuelle Tabelle (entsprechend der Klasse) ausgewertet, und anschließend wird aus der Tabelle die Adresse der virtuellen Methode verwendet.

Hierzu sollen nochmals unsere Klassen `Gegenstand` und `Buch` zum Einsatz kommen, die die virtuelle Methode `print()` verwenden. Im Beispiel wurde außerdem in der Klasse `Gegenstand` eine weitere virtuelle Methode `input()` hinzugefügt, die aber nicht in der abgeleiteten Klasse `Buch` redefiniert wird. Hier zunächst die neue virtuelle Methode `input()` der Klasse `Gegenstand`:

```
// gegenstand.h
#include <iostream>
#include <cstring>
#ifndef _GEGENSTAND_H_
#define _GEGENSTAND_H_
using namespace std;

class Gegenstand {
private:
   ...
   // alles wie gehabt

public:
   ...
   // alles wie gehabt
   ...
   virtual void input() {
      cout << "Artikel   : "; cin.getline(bezeichnung, 50);
      cout << "Anzahl    : "; cin >> anzahl;
      cout << "Nummer    : "; cin >> nummer;
      cout << "Preis     : "; cin >> preis;
      // Eingabepuffer leeren
      cin.clear();
      cin.ignore(cin.rdbuf()->in_avail());
      cin.get();
   }
};
#endif
```

Jetzt die Hauptfunktion:

```
// main.cpp
#include "gegenstand.h"
#include "buch.h"
int main( void ) {
// Basisklassenzeiger
   Gegenstand* gegenstandPtr;
   Gegenstand gegenstand1("Roman", 100, 124, 0),
              gegenstand2;
   Buch buch1( "IT-Fachbuch", 100, 123, 0,
               "C++ von A bis Z", "J. Wolf", 1000 );

   // gegenstand1 ausgeben mit Gegenstand::print
   gegenstandPtr =& gegenstand1;
   gegenstandPtr->print();

   // gegenstand2 einlesen mit Gegenstand::input
   gegenstandPtr =& gegenstand2;
   gegenstandPtr->input();

   // buch1 ausgeben mit Buch::print
   gegenstandPtr =& buch1;
   gegenstandPtr->print();
   cout << '\n';

   // buch1 einlesen mit (!) Gegenstand::input
   gegenstandPtr->input();
   cout << '\n';
   // verändertes buch1 ausgeben mit Buch::print
   gegenstandPtr->print();
   return 0;
}
```

Das Programm bei der Ausführung:

```
Artikel   : Roman
Anzahl    : 100
Nummer    : 124
Preis     : 0

Artikel   : Spielzeug
Anzahl    : 1
Nummer    : 125
Preis     : 100

Buchtitel : C++ von A bis Z
Autor     : J. Wolf
```

```
Seiten    : 1000
Artikel   : IT-Fachbuch
Anzahl    : 100
Nummer    : 123
Preis     : 0

Artikel   : IT-Fachbuch (C++)
Anzahl    : 100
Nummer    : 123
Preis     : 50

Buchtitel : C++ von A bis Z
Autor     : J. Wolf
Seiten    : 1000
Artikel   : IT-Fachbuch (C++)
Anzahl    : 100
Nummer    : 123
Preis     : 50
```

Zu diesem Programmablauf soll die folgende Abbildung (4.6) den Vorgang zwischen den virtuellen Zeigern und Tabellen demonstrieren.

In dieser Abbildung können Sie jeweils die virtuelle Tabelle der Klassen `Gegenstand` und `Buch` erkennen. Das Objekt `gegenstand1` ruft (über einen Basisklassenzeiger) die virtuelle Methode `print` aus der virtuellen Methodentabelle der Klasse `Gegenstand (Gegenstand::print)` auf. Dasselbe gilt für das Objekt `gegenstand2`, nur dass hierbei die virtuelle Methode `input()` der Klasse `Gegenstand (Gegenstand::input)` aufgerufen wird. Das Objekt `buch1` hingegen ruft die virtuelle Methode `print()` der Klasse `Buch (Buch::print)` aus der virtuellen Methodentabelle auf. Da es für die Klasse `Buch` keine eigene virtuelle `input`-Methode gibt, diese aber weitervererbt wurde, befindet sich in der virtuellen Methodentabelle der Klasse `Buch` ebenfalls ein Eintrag für die Methode `input()` der Klasse `Gegenstand (Gegenstand::input)`.

Bei beiden Klassen, `Gegenstand` und `Buch`, sind zwei Methoden in der virtuellen Tabelle eingetragen, wobei die Einträge für die Methode `input()` dieselben sind.

Nachteile der dynamischen Bindung

Natürlich gibt es in gewisser Hinsicht auch Nachteile der dynamischen Bindung gegenüber der statischen Bindung. Auf der einen Seite kann sich die Laufzeit der Anwendung verschlechtern, weil statt einer direkten Adressierung der Methoden im Maschinencode zwei Zeiger dereferenziert werden müssen. Auf der anderen Seite ist die Dereferenzierung von zwei Zeigern immer noch schneller als die Verwendung der `switch`-Statements und mehrerer `case`-Fallunterscheidungen.

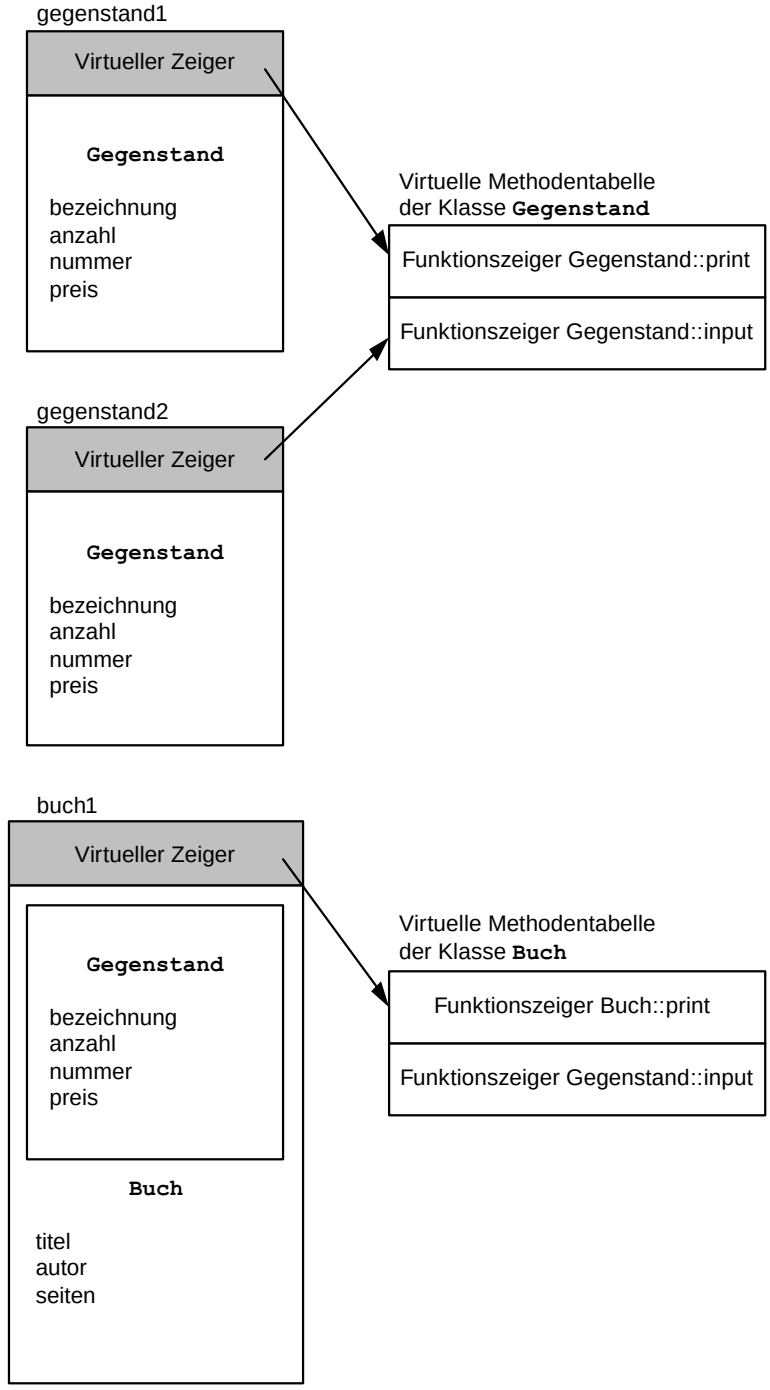

Abbildung 4.6 Virtuelle Methodenzeiger und Tabellen

Einzig den etwas höheren Speicherbedarf der Anwendung kann man nicht widerlegen. Dieser erhöhte Bedarf ergibt sich dadurch, dass die virtuellen Methodentabellen und die Zeiger für jedes einzelne Objekt Platz benötigen. Aber wenn man Anwendungen schreiben muss, die sehr schlank sein sollen, ist C++ sowieso nicht unbedingt die erste Wahl, und man kann dann auch auf C zurückgreifen.

4.8.5 Virtuelle Destruktoren bzw. Destruktoren abgeleiteter Klassen

Es wurde bereits erwähnt, wie der Vorgang bei den Destruktoren bei abgeleiteten Klassen vor sich geht. Wird ein Objekt erzeugt, bei dem mehrere Konstruktoren aufgerufen wurden, so werden die Destruktoren in der umgekehrten Reihenfolge ausgeführt. Bei Objekten abgeleiteter Klassen bedeutet dies, dass immer zuerst der Destruktor für die abgeleitete Klasse aufgerufen wird und anschließend der Destruktor für die Basisklasse. Im Grunde müssen Sie hierbei fast nichts machen, da Ihnen der Compiler diese Arbeit häufig abnimmt.

Aber auch hier entstehen Probleme, wenn in der abgeleiteten Klasse ein Objekt dynamisch erzeugt wird. Hierzu sei folgendes Beispiel gegeben:

```
// virtual5.cpp
#include <iostream>
#include <cstring>
using namespace std;

class Basisklasse {
public:
   Basisklasse() {
      cout << "Konstruktor Basisklasse\n";
   }
   ~Basisklasse() {
      cout << "Destruktor Basisklasse\n";
   }
};

class abgeleiteteKlasse : public Basisklasse {
private:
   char *Zkette;

public:
   abgeleiteteKlasse( const char *str = "" ) {
      cout << "Konstruktor abgeleiteteKlasse\n";
      Zkette = new char [strlen(str)+1];
```

```
      strcpy( Zkette, str );
   }
   ~abgeleiteteKlasse() {
      cout << "Destruktor abgeleiteteKlasse\n";
      delete [] Zkette;
   }
};

int main( void ) {
   Basisklasse* basisPtr;
   basisPtr = new abgeleiteteKlasse("Zeichenkette");
   delete basisPtr;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Konstruktor Basisklasse
Konstruktor abgeleiteteKlasse
Destruktor Basisklasse
```

Hier fällt gleich auf, dass der Destruktor für die abgeleitete Klasse nicht aufgerufen wird. Das Problem ist, dass der Compiler das durch `basisPtr` adressierte Objekt nicht kennt, weil `basisPtr` ein Zeiger der Basisklasse ist. Daher wird auch der Destruktor für die Basisklasse aufgerufen. Hätten Sie außerdem in diesem Beispiel keinen Destruktor der Klasse `abgeleiteteKlasse` definiert, so hätte dies keine problematischen Folgen, weil dann der Destruktor der Klasse `abgeleitete-Klasse` keine Arbeit ausführt. Aber hier wurde der Destruktor definiert und muss daher explizit aufgerufen werden.

Das Problem lässt sich ganz einfach dadurch lösen, dass man auch die Destruktoren virtuell deklariert. Im Beispiel müssen Sie daher nur den Destruktor der Basisklasse als virtuell deklarieren, und schon ist auch der Destruktor der abgeleiteten Klasse implizit virtuell.

```
class Basisklasse {
public:
   Basisklasse() {
      cout << "Konstruktor Basisklasse\n";
   }
   virtual ~Basisklasse() {
      cout << "Destruktor Basisklasse\n";
   }
};

class abgeleiteteKlasse : public Basisklasse {
```

```
private:
   char *Zkette;

public:
   abgeleiteteKlasse( const char *str = "" ) {
      cout << "Konstruktor abgeleiteteKlasse\n";
      Zkette = new char [strlen(str)+1];

      strcpy( Zkette, str );
   }
   // Jetzt, implizit virtuell
   ~abgeleiteteKlasse() {
      cout << "Destruktor abgeleiteteKlasse\n";
      delete [] Zkette;
   }
};
```

Nur das Schlüsselwort `virtual` in der Basisklasse sorgt dafür, dass wie bei den Methoden jetzt auch der richtige Destruktor aufgerufen wird. So sieht die Ausführung des Programms *virtual5.cpp* nach dieser kleinen Änderung wie folgt aus:

```
Konstruktor Basisklasse
Konstruktor abgeleiteteKlasse
Destruktor abgeleiteteKlasse
Destruktor Basisklasse
```

Hinweis [«]

Folgendes sollten Sie sich daher zur Regel machen: Wenn eine Basisklasse anderen Klassen dienen soll, sollte diese immer einen virtuellen Destruktor haben. Selbst wenn die Basisklasse keinen eigenen Destruktor benötigt, sollten Sie zumindest einen virtuellen Dummy-Destruktor deklarieren.

Virtuelle Konstruktoren?

Im Gegensatz zu Destruktoren können Konstruktoren niemals virtuell sein. Ein Konstruktor dient schließlich dazu, ein Objekt mit einem bestimmten Typ zu initialisieren. Der Typ ist dabei immer bekannt und daher statisch. Da die Objekte nur auf Veranlassung des Programms erzeugt werden, wird dabei auch der exakte Typ angegeben.

4.8.6 Polymorphismus und der Zuweisungsoperator

Wenn Sie einen Zuweisungsoperator als virtuell definieren wollen, sollten Sie vorsichtig sein, weil eine solche Zuweisung nicht immer polymorph ist. Dies ist schon deswegen nicht immer möglich, weil polymorphe Funktionen ja stets dieselbe Signatur haben müssen. Ein Beispiel, was hiermit gemeint ist:

```
// virtual6.cpp
#include <iostream>
#include <cstring>
using namespace std;


class Basisklasse {
public:
    virtual Basisklasse &operator=(const Basisklasse &b) {
      return *this;
    }
};

class abgeleiteteKlasse : public Basisklasse {
public:
   int iwert;
   abgeleiteteKlasse() : iwert(0) { }
   abgeleiteteKlasse &operator=(const abgeleiteteKlasse &a){
     iwert = a.iwert;
     return *this;
   }
};

int main( void ) {
   Basisklasse *basisPtr;
   abgeleiteteKlasse aObjekt1, aObjekt2;
   aObjekt1.iwert = 1111;
   basisPtr =& aObjekt2;
   // aObjekt2 mit dem Wert von aObjekt1 belegen
   *basisPtr = aObjekt1;
   // Ausgabe nicht wie erwartet !!!
   cout << aObjekt2.iwert << '\n';
   return 0;
}
```

Statt der Ausgabe des Wertes `1111`, wie hier beabsichtigt, wird `0` ausgegeben. Das Problem liegt darin, dass der Zuweisungsoperator in der abgeleiteten Klasse einen anderen Typ hat als der Zuweisungsoperator in der Basisklasse. Und daher stimmt die Signatur der beiden Zuweisungsoperator-Methoden nicht überein, und sie sind somit ohne jeden Bezug zueinander. Sie haben zwar mit `Basisklasse::operator=(const Basisklasse&)` eine virtuelle Methode definiert, aber diese wird in der Methode `abgeleiteteKlasse` nicht redefiniert. Würde die Methode in der abgeleiteten Klasse folgenden Funktionskopf haben

```
abgeleiteteKlasse &operator=(const Basisklasse &a)
```

dann hätten Sie einen echten polymorphen Zuweisungsoperator (ob dieser hier Sinn macht, sei dahingestellt).

4.8.7 Rein virtuelle Methoden und abstrakte Basisklassen

Damit virtuelle Methoden über die Basisklassen-Schnittstelle auch in der abgeleiteten Klasse zur Verfügung stehen, müssen diese stets in der Basisklasse deklariert sein. Auch wenn in der Basisklasse keine Aufgabe für diese virtuelle Methode vorhanden ist, ist eine Deklaration einer leeren Dummy-Methode notwendig, was zwar Speicherplatz verbraucht, aber die einzige Möglichkeit ist, um die virtuellen Methoden in der abgeleiteten Klasse zu nutzen. Virtuelle Methoden verwenden zudem auch Speicherplatz in der virtuellen Tabelle.

In C++ haben Sie die Möglichkeit, eine Methode als eine *rein virtuelle* Methode zu deklarieren. Eine solche Deklaration sieht wie folgt aus:

```
class Basisklasse {
public:
   virtual void funktion() = 0;
};
```

Durch die Deklaration der Methode und das Anhängen von `=0` erhält der Compiler die Anweisung, dass in dieser Klasse keine Definition der Methode vorhanden sein muss. In der virtuellen Tabelle wird daher für ein Objekt dieser Klasse `NULL` eingetragen. Rein virtuelle Methoden werden in der Praxis häufig auch als *abstrakte Methoden* bezeichnet. Solche abstrakten Methoden haben praktisch in der Klasse, in der diese deklariert werden, keine Bedeutung, sie werden aber in einer abgeleiteten Klasse definiert:

```
class Desktop {
   public:
   Desktop() {}
   virtual ~Desktop() {}
   virtual long get_X() = 0;
   virtual long get_Y() = 0;
   virtual void Zeichnen() = 0;
};
```

In dieser Klasse `Desktop` werden mehrere virtuelle Methoden deklariert. Sie dient als Basisklasse zur Ableitung anderer Klassen, wie zum Beispiel ein einfaches Fenster mit Rollbalken, das auf dem Desktop angezeigt werden kann, oder auch eine Nachrichtenbox – ein kleiner Window-Manager eben.

Abstrakte Klassen

Natürlich sollte auch klar sein, dass von Klassen, die rein virtuelle Methoden enthalten, keine Objekte erzeugt werden können. Das Objekt würde sonst versuchen, eine Methode aufzurufen, die es gar nicht gibt. Bezogen auf die Klasse `Desktop`, ist Folgendes nicht möglich:

```
Desktop ein_Desktop; // falsch !!!
```

Solche Klassen, die mindestens eine rein virtuelle Methode enthalten und von denen kein Objekt erzeugt werden kann, werden als *abstrakte (Basis-)Klasse* (oder auch *abstrakte Datentypen*) bezeichnet.

Abstrakte Klassen (bzw. auch abstrakte Datentypen) stellen in C++ immer ein Konzept und kein Objekt dar. Somit ist eine abstrakte Klasse immer die Basisklasse für andere Klassen. Sobald Sie also eine rein virtuelle Methode wie folgt deklarieren

```
virtual void funktion() = 0;
```

signalisieren Sie damit:

- Es kann kein Objekt dieser Klasse erzeugt werden.
- Sie wollen einen abstrakten Typ erstellen, um eine gemeinsame Funktionalität für mehrere (abgeleitete) Klassen bereitzustellen.
- Diese Methode soll auf jeden Fall redefiniert werden.

Auch wenn es hierbei kein Objekt einer abstrakten Klasse gibt, können Sie wieder einen entsprechenden Zeiger bzw. eine Referenz definieren und verwenden:

```
Desktop* deskPtr;
```

Mit Hilfe dieses Zeigers (bzw. dieser Referenz) kann dann wieder auf Objekte der abgeleiteten Klasse gezeigt werden.

Bevor auch die Erklärungen hier zu abstrakt werden, soll ein Beispiel zur Demonstration erstellt werden. Das Beispiel zeigt eine geschlossene Hierarchie von Klassen. Zunächst erzeugen wir eine abstrakte Basisklasse `Desktop` und leiten dann davon die Klassen `Fenster` und `Box` ab. Hier nochmals die abstrakte Klasse `Desktop`:

```
class Desktop {
public:
   Desktop() {}
   virtual ~Desktop() {}
   virtual long get_X() = 0;
   virtual long get_Y() = 0;
   virtual void Zeichnen() = 0;
};
```

Alle aus dieser abstrakten Basisklasse abgeleiteten Klassen erben jetzt die rein virtuellen Methoden in der »nackten« Form. Wenn Sie nicht wollen, dass die abgeleitete Klasse auch abstrakt ist, müssen Sie alle drei Methoden redefinieren, um Objekte davon zu erzeugen. Wenn also eine abgeleitete Klasse `Fenster` von der Klasse `Desktop` erbt und `Desktop` wie hier drei rein virtuelle Methoden enthält, müssen Sie in `Fenster` alle drei Methoden redefinieren, oder `Fenster` bleibt ebenfalls eine abstrakte Klasse.

Die Bedeutung des folgenden Beispiels lässt sich schnell erklären. Auf einem »Desktop«, der nicht als Objekt instantiiert werden darf, weil er ja schon existiert (hier: auf dem Bildschirm dargestellt wird), soll entweder ein einfaches »Fenster« oder eine »Box« (Nachrichtenbox) dargestellt werden. Um zu verhindern, dass nicht doch ein Objekt vom Typ `Desktop` erzeugt wird, erstellen wir diese Klasse so, dass sie nur als Schnittstelle für davon abgeleitete Klassen funktioniert. Dies erledigt man mit einer abstrakten Basisklasse. In der abgeleiteten Klasse redefinieren Sie nun die Methoden `get_X()`, `get_Y()` und `Zeichnen()`.

```
// virtual7.cpp
#include <iostream>
#include <cstring>
using namespace std;

class Desktop {
public:
   Desktop() {}
   virtual ~Desktop() {}
   virtual long get_X() = 0;
   virtual long get_Y() = 0;
   virtual void Zeichnen() = 0;
};

class Fenster : public Desktop {
private:
   long x;
   long y;
public:
   Fenster( long xlen, long ylen ) : x(xlen), y(ylen) {}
   ~Fenster() { cout << "Fenster zerstört\n"; }
   long get_X() { return x; }
   long get_Y() { return y; }
   void Zeichnen() {
      cout << "Fenster (" << x << "*" << y << ") "
           << "gezeichnet\n";
   }
};
```

```
class Box : public Desktop {
private:
   long pos_x;
   long pos_y;
public:
   Box( long x, long y) : pos_x(x), pos_y(y) {}
   ~Box() { cout << "Nachrichten-Box zerstört\n"; }
   long get_X() { return pos_x; }
   long get_Y() { return pos_y; }
   void Zeichnen() {
      cout << "Nachrichten-Box an Position (x:" << pos_x
           << "/y:" << pos_y << ") gezeichnet\n";
   }
};

int main( void ) {
   Desktop* deskPtr1;
   Desktop* deskPtr2;
   deskPtr1 = new Fenster(800, 600);
   deskPtr2 = new Box(30, 30);
   deskPtr1->Zeichnen();
   deskPtr2->Zeichnen();
   delete deskPtr1;
   delete deskPtr2;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Fenster (800*600) gezeichnet
Nachrichten-Box an Position (x:30/y:30) gezeichnet
Fenster zerstört
Nachrichten-Box zerstört
```

Zusammengefasst kann man sagen, dass abstrakte Klassen eine polymorphe Schnittstelle für abgeleitete Klassen darstellen. Damit lässt sich eine allgemeine Funktionalität zunächst als rein virtuelle Funktionen implementieren, die dann über Zeiger oder Referenzen auf die abstrakte Klasse aufgerufen werden. Wird in der abgeleiteten Klasse eine Methode aus der rein virtuellen Klasse redefiniert, wird diese auch ausgeführt.

Konstruktoren abstrakter Klassen

Abstrakte Klassen benötigen immer einen Konstruktor – auch wenn diese selbst keine Objekte erzeugen. Aber bei jedem Objekt einer abgeleiteten Klasse wird auch der Konstruktor der Basisklasse aufgerufen, der für die Initialisierung der Basiseigenschaften (falls vorhanden) zuständig ist.

Zuweisungsoperator und Kopierkonstruktor abstrakter Klassen

Solange Sie keine dynamischen Elemente in der abstrakten Klasse verwenden, müssen Sie nicht explizit einen Kopierkonstruktor oder eine Zuweisung definieren, und Sie können die Standardversionen verwenden, die automatisch zur Verfügung gestellt werden. Befinden sich allerdings dynamische Eigenschaften in der Basisklasse, so müssen Sie einen eigenen Kopierkonstruktor und eine Zuweisung definieren.

Es ist nicht immer nötig, in abgeleiteten Klassen einen eigenen Kopierkonstruktor zu definieren, auch hier besteht erst Bedarf, sobald die abgeleitete Klasse ein dynamisches Element besitzt (siehe auch Abschnitt 4.4.6, »Dynamische Klassenelemente«).

4.8.8 Probleme mit der Vererbung und der dynamic_cast-Operator

Ein Problem, auf das vor allem die noch unerfahrenen Programmierer stoßen könnten, entsteht dann, wenn man eine Methode in der abgeleiteten Klasse hinzufügen möchte, diese aber überhaupt nicht zur Basisklasse passt – beispielsweise, wenn man in der abgeleiteten Klasse `Fenster` vom Listing *virtual7.cpp* folgende Methode hinzufügt:

```
class Fenster : public Desktop {
private:
   long x;
   long y;
public:
   Fenster( long xlen, long ylen ) : x(xlen), y(ylen) {}
   ~Fenster() { cout << "Fenster zerstört\n"; }
   long get_X() { return x; }
   long get_Y() { return y; }
   void Zeichnen() {
      cout << "Fenster (" << x << "*" << y << ") "
           << "gezeichnet\n";
   }
   void Neuzeichnen( long xlen, long ylen ) {
      x=xlen;
      y=ylen;
      // Neu zeichnen
      Zeichnen();
   }
};
```

Wenn Sie hier die Methode `Neuzeichnen` mit dem Basisklassenzeiger verwenden, erhalten Sie eine Fehlermeldung vom Compiler, dass die Methode keine der

Basisklassen (hier `Desktop`) ist. Der Compiler kann also beim Auflösen der virtuellen Tabelle keinen solchen Eintrag finden.

Um es gleich zu sagen: Dieses Beispiel entspricht einem schlecht überdachten Entwurf des Programmierers. Denn wenn man schon einen Zeiger auf eine Basisklasse verwenden will, um auf Objekte abgeleiteter Klassen zuzugreifen, will man dies gewöhnlich auch polymorph tun.

Wie kann man auf diese Funktion zugreifen, ohne gleich das ganze Programm umzuschreiben? Zwar könnten Sie jetzt diese Methode in die Basisklasse schieben, wenn Sie allerdings zur Basisklasse schon eine ganze Sammlung abgeleiteter Klassen hinzugefügt haben, müssen Sie alle anderen Klassen gegebenenfalls anpassen und testen. Außerdem wird dadurch die Basisklasse unnötig groß und die Wartung der abgeleiteten Klasse verkompliziert.

Ein zweiter Weg wäre, den Zeiger der Basisklasse oder der Referenz in einen Zeiger oder eine Referenz der abgeleiteten Klasse umzuwandeln. Da wir hier eine dynamische Bindung haben, benötigen wir auch eine dynamische Umwandlung. Für diesen Zweck können Sie den `dynamic_cast<>()`-Operator verwenden (siehe Abschnitt 3.7.2, »Explizite Typumwandlung«).

Damit wird der Basisklassenzeiger ebenfalls erst zur Laufzeit überprüft. Wenn die Umwandlung erfolgreich war, wird ein einwandfreier Zeiger der abgeleiteten Klasse zurückgegeben. Gibt es kein Objekt der abgeleiteten Klasse oder scheitert die Umwandlung, wird `0` bzw. `NULL` zurückgegeben. Soll also die Methode `Neuzeichnen` in der abgeleiteten Klasse `Fenster` ausgeführt werden, müssen Sie wie folgt vorgehen:

```
...
int main( void ) {
   Desktop* deskPtr1;
   Fenster* fensterPtr;
   deskPtr1 = new Fenster(800, 600);
   deskPtr1->Zeichnen();
   fensterPtr = dynamic_cast<Fenster*> (deskPtr1);
   if(fensterPtr != NULL) {
      fensterPtr->Neuzeichnen(640, 480);
   }
   // Der Beweis ...
   cout << "Neue Fenstergröße : " << deskPtr1->get_X()
        << "x" << deskPtr1->get_Y() << "\n";

   delete deskPtr1;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Fenster (800*600) gezeichnet
Fenster (640*480) gezeichnet
Neue Fenstergröße : 640x480
Fenster zerstört
```

Um an das `Fenster`-Objekt heranzukommen und die Methode `Neuzeichnen` aufzurufen, erzeugt man zunächst einen `Fenster`-Zeiger und nimmt eine Umwandlung mit dem `dynamic_cast`-Operator vor. Ist der Rückgabewert dieser Umwandlung ungleich `0` bzw. ungleich `NULL`, können Sie auf die Methode `Neuzeichnen` zugreifen.

4.8.9 Fallbeispiel: Verkettete Listen

Alles bisher Gelernte lässt sich nun hervorragend an einem Beispiel mit verketteten Listen demonstrieren. Heute wird zwar keiner mehr verkettete Listen selbst implementieren und auf eine der vielen Bibliotheken (z. B. STL) zurückgreifen, wenn man sich aber ernsthaft mit C++ befassen muss, sind verkettete Listen Pflicht.

Wer bereits Erfahrungen mit den verketteten Listen in C gemacht hat und meint, er könne diesen Teil überfliegen, dem sei gesagt, dass die OOP-Implementierung der verketteten Listen nicht mehr viel mit der prozeduralen Programmierung gemein hat.

Wer noch nie etwas von verketteten Listen gehört hat, erhält hier jetzt eine kurze Erklärung, worum es sich dabei handelt. Nehmen Sie zunächst ein Array. In einem Array können Sie eine feste Anzahl von Elementen eines bestimmten Typs speichern. Manchmal ist es aber ein Hindernis, wenn man auf eine feste Größe fixiert ist.

Abbildung 4.7 Interne Ansicht eines Arrays

Auch Arrays lassen sich mit einen gewissen Aufwand dynamisch implementieren, aber das Kopieren hat bei sehr großem Datenaufkommen (z. B. mehrere GB) wohl wenig Sinn und birgt zudem einige Risiken. Wie machen das also die großen Datenbanken wie MySQL und Co.? Die Antwort gleich vorweg: Große Datenbanken verwenden Bäume, was aber auch eine Form von Liste ist. Somit basiert zunächst alles auf der einfachsten Grundlage, den einfach verketteten Listen:

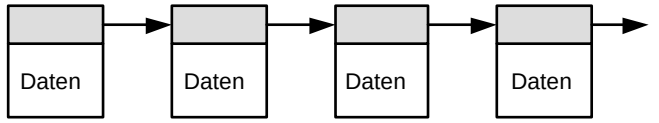

Abbildung 4.8 Einfach verkettete Listen

Bei den verketteten Listen handelt es sich um eine einfache Datenstruktur mit Elementen beliebigen Typs, die aneinandergehängt werden. Dadurch soll – gemäß OOP – eine Klasse erstellt werden, die das Objekt der Daten darstellt und gleichzeitig auch auf das nächste Objekt vom selben Typ zeigt. Sie erzeugen praktisch mit jeder Instanz ein neues Objekt und hängen dieses in einer Liste an das andere, bei Bedarf auch sortiert.

Bei einer einfach verketteten Liste beginnt man am Anfang der Liste und sucht nach einem bestimmten Knoten, an dem das Element eingefügt werden soll. Dabei durchläuft man die Liste Knoten für Knoten bis zum letzten Knoten. Neben einfach verketteten Listen gibt es natürlich noch weitere und komplexere Listenstrukturen, wobei die doppelt verketteten Listen und die binären Bäume ebenfalls zu den grundlegenden Datenstrukturen gehören (siehe Abbildungen 4.9 und 4.10).

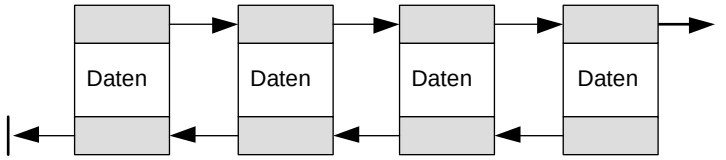

Abbildung 4.9 Doppelt verkettete Listen

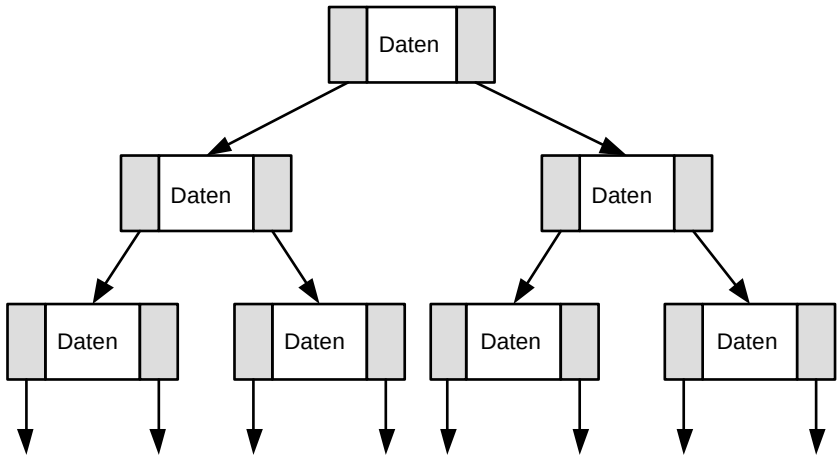

Abbildung 4.10 Binäre Bäume

Die einzelnen Klassen (Komponenten)

Unabhängig von der verketteten Liste benötigen Sie zunächst eine Klasse, in der Sie die Daten speichern wollen, die anschließend in der Kette aneinandergereiht werden sollen. In diesem Beispiel begnügen wir uns mit einer einfachen Klasse, die lediglich eine Eigenschaft `iwert` besitzt, die diese Klasse aufnehmen kann. Diese Klasse bekommt außerdem zwei Methoden – eine, mit der zwei Objekte von der Klasse `Daten` miteinander verglichen werden können, und eine Methode zum Ausgeben dieser Daten auf dem Bildschirm. Hier die komplette Klasse:

```
// daten.h
#include <iostream>
using namespace std;
#ifndef _DATEN_H_
#define _DATEN_H_

class Daten {
private:
   int iwert;
public:
   // Konstruktor
   Daten( int iVal ): iwert(iVal) {
   // Zu Debug- bzw. Verständniszwecken ggf. entfernen
   cout << "Objekt [Daten] erzeugt\n";
   }
   // Destruktor
   ~Daten( ) {}
   int vergleichen( const Daten& );
   void anzeigen() const;
};

// Zum Vergleichen der Eigenschaften zweier Objekte
int Daten::vergleichen( const Daten& d ) {
   // Größer
   if( iwert > d.iwert ) { return 1; };
   // Kleiner
   if( iwert < d.iwert ) { return -1;  };
   // Gleich
   return 0;
}

void Daten::anzeigen() const {
   cout << iwert << '\n';
}
#endif
```

Durch eine zusätzliche Klasse für die Daten können Sie schon erkennen, dass hier nicht die Daten verkettet werden, sondern die gleich erzeugten Knoten. In der prozeduralen Programmierung sind dies gewöhnlich die Daten, die aneinandergehängt werden.

Für den oder die Knoten könnten wir je eine Klasse für den Anfang der Liste und eine für das Ende der Liste schreiben. Natürlich benötigen Sie auch einen Knoten, der die Daten verwaltet und behandelt. Damit wir nicht alles in dreifacher Ausführung schreiben müssen, verwenden wir als Basisklasse einen abstrakten (rein virtuellen) Typ als Knoten. Somit haben Sie zunächst vier Klassen für den Knoten, eine abstrakte und drei abgeleitete Klassen:

- *Knoten* – die abstrakte Basisklasse mit den Methoden `einfuegen()` und `anzeigen()`, die von allen abgeleiteten Klassen ebenfalls redefiniert werden müssen.
- *AnfangsKnoten* – dieser Knoten nimmt keine Daten auf, sondern sorgt dafür, dass alle Daten, die kommen, hinter diesem Knoten eingefügt werden. Vor diesem Knoten kommt nichts.
- *AllgemeinerKnoten* – der Knoten nimmt die eigentlichen Daten auf und fügt diese in der Kette an einer entsprechenden Stelle ein. Hier ist auch die Kernmethode `einfuegen()` redefiniert, die ermittelt, wo dieser Knoten eingefügt wird.
- *EndKnoten* – dieser Knoten sorgt dafür, dass keine Daten an ihm vorbeikommen, und dient als Ende-Markierung. Kein *AllgemeinerKnoten* wird hinter diesem Knoten eingefügt.

Somit sind die Klassen `AnfangsKnoten` und `EndKnoten` ein fester Knoten, der sich, einmal erzeugt, nicht mehr verändert. Zwischen diesen beiden Knoten wird ein Objekt der Klasse `AllgemeinerKnoten` sortiert eingefügt – und `AllgemeinerKnoten` verwaltet auch die Klasse `Daten`.

Da die Klasse `Knoten` selbst nur die abstrakte Basisklasse darstellt, soll diese hier vorgestellt werden:

```
// Knoten ist die abstrakte Basisklasse
// Alle abgeleiteten Klassen müssen "einfuegen"
// und "anzeigen" redefinieren
class Knoten {
public:
   Knoten() {}
   virtual ~Knoten() {}
   virtual Knoten* einfuegen( Daten* d ) = 0;
   virtual void anzeigen() = 0;
};
```

Dem Ganzen setzen wir noch eine »Maske« auf und schreiben eine weitere Klasse `Liste`, die die ganze Knotengeschichte vor dem Anwender dieser Klasse verbirgt. Natürlich benötigt diese Klasse mindestens ein Knotenelement und die Methoden `anzeigen()` und `einfuegen()`, die allerdings nichts mit den gleichnamigen Methoden der Knotenklassen zu tun haben. Dennoch werden über das Knotenelement der Klasse `Liste` durch das Aufrufen der Methoden `einfuegen()` bzw. `anzeigen()` tatsächlich die eigentlichen Knotenmethoden aufgerufen. Hier also zunächst die Klasse `Liste`:

```
// ------ Unabhängige Klasse Liste -------------------
class Liste {
private:
   AnfangsKnoten *anfang;
public:
   // Bei Anlegen gleich ein Objekt "Anfangsknoten" erzeugen
   // der Konstruktor von "AnfangsKnoten" erzeugt wiederum
   // ein Objekt "EndKnoten", auf das dieser gleich zeigt.
   Liste() { anfang = new AnfangsKnoten; }
   ~Liste() { delete anfang; }
   // An die Methode einfuegen() von Knoten weiterleiten
   void einfuegen( Daten* d ) {
      anfang->einfuegen(d);
   }
   // An die Methode anzeigen() von Knoten weiterleiten
   void alles_anzeigen() {
      anfang->anzeigen();
   }
};
```

Im Hauptprogramm werden Sie ausschließlich mit Objekten vom Typ `Liste` zu tun haben. Wenn Sie ein Objekt davon anlegen, erzeugt der Konstruktor auch ein Objekt vom Typ `AnfangsKnoten` (beim Beenden wird es vom Destruktor auch wieder abgebaut).

Um den Vorgang besser zu verstehen, wollen wir einen Testlauf machen. Daher zunächst die Hauptfunktion:

```
// main.cpp
#include "daten.h"
#include "lliste.h"

int main( void ) {
   Liste elemente;
   Daten* daten;
   int iwert;
   for(;;) {
```

```
      cout << "Wert eingeben (0=Ende): ";
      // Falsche Eingabe oder 0
      if( (!(cin >> iwert)) || iwert ==0 )
         break; // Ende
      daten = new Daten(iwert);
      elemente.einfuegen(daten);
   }
   elemente.alles_anzeigen();
   return 0;
}
```

Sie sehen hier keine Spur von den einzelnen Knoten, Sie finden nur die Klassen `Daten` und `Liste` wieder.

Wenn also ein Objekt vom Typ `Liste` erzeugt wurde, wird durch den Konstruktor ein Objekt vom Typ `AnfangsKnoten` erzeugt. Hierzu die abgeleitete Klasse `AnfangsKnoten`:

```
// ------ Abgeleitete Klasse AnfangsKnoten ----------------
// Knoten, der "nur" immer auf das erste Element
// der Liste zeigt
class AnfangsKnoten : public Knoten {
private:
   // ... zeigt immer auf das erste Element
   Knoten* next;
public:
   // Konstruktor
   AnfangsKnoten() {
      // ... gleich auch einen Endknoten erzeugen
      next = new EndKnoten;
      // Zu Debug- bzw. Verständniszwecken ggf. entfernen
      cout << "Objekt [AnfangsKnoten] erzeugt\n";
   }
   ~AnfangsKnoten() { delete next; }
   // Implizit virtual
   Knoten* einfuegen( Daten* d );
   // Implizit virtual
   void anzeigen();
};

Knoten* AnfangsKnoten::einfuegen( Daten* d ) {
   // Am Anfang kommen keine Daten rein, daher an den
   // nächsten Knoten weiterreichen
   next = next->einfuegen(d);
   return this;
}
```

```
void AnfangsKnoten::anzeigen() {
   next->anzeigen();
}
```

Die Klasse `AnfangsKnoten` ruft zunächst auch wieder nur den Konstruktor auf. Dieser erzeugt hier ein Objekt vom Typ `EndKnoten`, auf den der `next`-Zeiger zunächst verweist. Somit zeigt das Objekt vom Typ `AnfangsKnoten` zum Start auf einen Typ vom Objekt `EndKnoten`. Hier die Klasse `EndKnoten`:

```
// ------ Abgeleitete Klasse EndKnoten -------------------
// Der EndKnoten dient als Endpunkt der Liste
class EndKnoten : public Knoten {
public:
   EndKnoten() {
      // Zu Debug- bzw. Verständniszwecken ggf. entfernen
      cout << "Objekt [EndKnoten] erzeugt\n";
   }
   ~EndKnoten() {}
   // Implizit virtual
   Knoten* einfuegen( Daten* d );
   // Implizit virtual
   void anzeigen() { };
};

// Daten werden immer vor dem Ende eingefügt
Knoten* EndKnoten::einfuegen( Daten* d ) {
   AllgemeinerKnoten* daten=new AllgemeinerKnoten(d, this);
   return daten;
}
```

Den momentanen Programmzustand kann man sich »bildlich« folgendermaßen vorstellen:

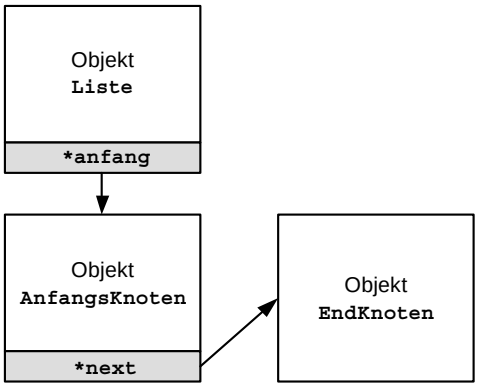

Abbildung 4.11 Momentaner Programmzustand

Bis jetzt haben Sie als Benutzer noch keinen Wert eingegeben. Nun werden Sie aber dazu aufgefordert (wenn Sie das Hauptprogramm ausführen):

```
Objekt [EndKnoten] erzeugt
Objekt [AnfangsKnoten] erzeugt
Wert eingeben (0=Ende): 123
Objekt [Daten] erzeugt
Objekt [AllgemeinerKnoten] erzeugt
Wert eingeben (0=Ende):
```

Es wurde also der Wert `123` eingegeben. Kurz darauf werden die Daten und ein allgemeiner Knoten erzeugt und das neue Element zur Liste hinzugefügt. Das Einfügen soll jetzt etwas genauer erläutert werden. Es wird in der Hauptfunktion wie folgt eingeleitet:

```
Liste elemente;
Daten* daten;
...
daten = new Daten(123);
elemente.einfuegen(daten);
```

Zunächst wird also die Methode `Liste::einfuegen()` mit dem `Daten`-Objekt aufgerufen.

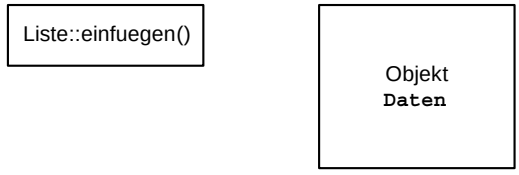

Abbildung 4.12 Aufruf der ersten Methode aus dem Hauptprogramm

Diese Methode leitet die Verantwortung an den `AnfangsKnoten` weiter bzw. an die Methode `AnfangsKnoten::einfuegen()`. Auch der `AnfangsKnoten` gibt seine Arbeit an den Knoten weiter, auf den der Zeiger `next` zeigt (was am Anfang noch der `EndKnoten` ist, auf den Sie beim Erzeugen eines `AnfangsKnotens` verwiesen haben – siehe Abbildung 4.13).

```
Knoten* AnfangsKnoten::einfuegen( Daten* d ) {
   // Am Anfang kommen keine Daten rein, daher an den
   // nächsten Knoten weiterreichen
   next = next->einfuegen(d);
   return this;
}
```

In der Methode `EndKnoten::einfuegen()` wird das übernommene Objekt direkt vor dem `EndKnoten` eingefügt. Deshalb wird ein neues Objekt vom Typ `Allge-`

meinerKnoten erzeugt. Damit wird auch gleich der Konstruktor AllgemeinerKnoten mit den Daten und einem next-Zeiger mit der Adresse des übergebenen Knotens aufgerufen. Am Anfang ist dies die Adresse des EndKnotens, da dieser seinen eigenen this-Zeiger übergeben hat (siehe Abbildung 4.14).

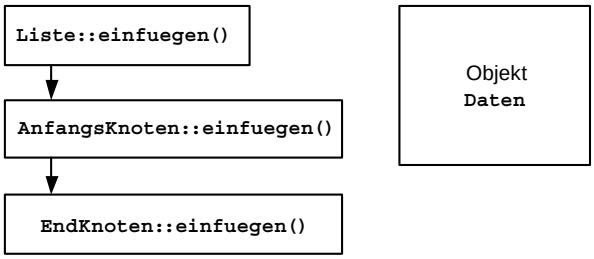

Abbildung 4.13 Nach weiteren Delegationen

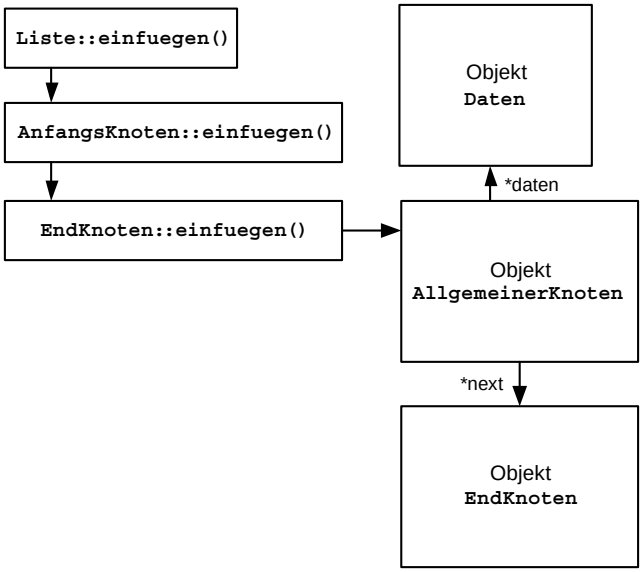

Abbildung 4.14 Am Ende angekommen ...

Hierzu die Klasse AllgemeinerKnoten:

```
// ------ Abgeleitete Klasse AllgemeinerKnoten -------------
// Diese Klasse ist für die eigentliche Verwaltung der Daten
// verantwortlich - im Beispiel wird zwar ein Objekt vom Typ
// "Daten" verwendet, aber mit Template (späteres Kapitel)
// können Sie hier noch eins drauf setzen und die Listen-
// Klasse verallgemeinern und somit (fast) unabhängig von
// den Daten machen
```

```
class AllgemeinerKnoten : public Knoten {
private:
   Daten* daten;
   Knoten* next;
public:
   // Konstruktor
   AllgemeinerKnoten( Daten* d, Knoten* n ):
                                    daten(d), next(n) {
      // Zu Debug- bzw. Verständniszwecken ggf. entfernen
      cout << "Objekt [AllgemeinerKnoten] erzeugt\n";
   }
   // Destruktor
   ~AllgemeinerKnoten() {
      delete next;
      delete daten;
   }
   // Implizit virtual
   Knoten* einfuegen( Daten* d );
   // Implizit virtual
   void anzeigen();
};

// Die wichtigsten Methoden in diesem Programm
Knoten* AllgemeinerKnoten::einfuegen( Daten* d ) {
   // Wir sortieren aufwärts - kleiner Wert vor großem Wert
   int ret = daten->vergleichen( *d );
   switch( ret ) {
      case 1: {
         // Neue Daten vor den aktuellen einordnen
         AllgemeinerKnoten* dKnoten =
            new AllgemeinerKnoten( d, this );
         return dKnoten;
      }
      case -1:
         // größer als das aktuelle Element -
         // weiter zum nächsten Knoten
         next = next->einfuegen( d );
         return this;
   }
   return this;
}

void AllgemeinerKnoten::anzeigen() {
   daten->anzeigen();
   next->anzeigen();
}
```

Wenn das Objekt `AllgemeinerKnoten` erzeugt wurde, wird diese Adresse an den Zeiger `daten` übergeben, zugewiesen und als Wert von der Methode `Endknoten::einfuegen()` zurückgegeben (siehe Abbildung 4.15):

```
// Daten werden immer vor dem Ende eingefügt
Knoten* EndKnoten::einfuegen( Daten* d ) {
   AllgemeinerKnoten* daten=new AllgemeinerKnoten(d, this);
   return daten;
}
```

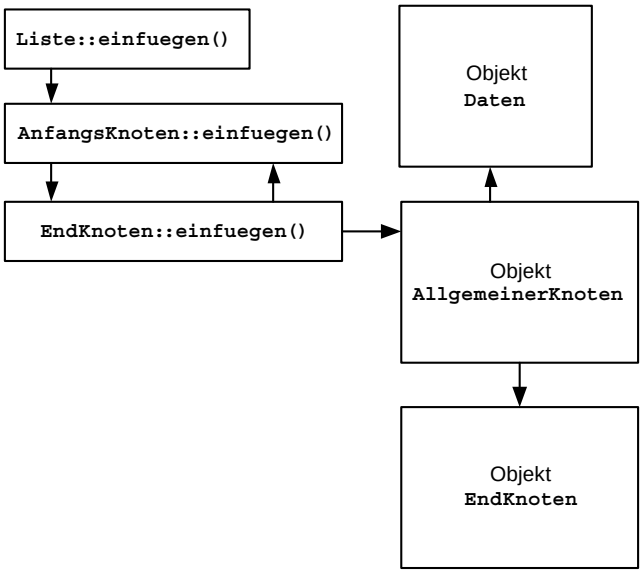

Abbildung 4.15 ... geht es wieder zurück ...

Den zurückgegebenen Wert von `EndKnoten::einfuegen()` erhält die Methode `AnfangsKnoten::einfuegen()`, in der die Adresse des `AllgemeinenKnotens` dem Zeiger `next` von `AnfangsKnoten` zugewiesen wurde:

```
Knoten* AnfangsKnoten::einfuegen( Daten* d ) {
   // Am Anfang kommen keine Daten rein, daher an den
   // nächsten Knoten weiterreichen
   next = next->einfuegen(d);
   return this;
}
```

Der Rückgabewert von `AnfangsKnoten::einfuegen()` wiederum wird am Ende an das Objekt vom Typ `Liste` zurückgegeben, wo die Adresse nicht mehr benötigt und verwendet wird, da wir ja bereits den `AnfangsKnoten` von der Liste haben (siehe Abbildung 4.16):

```
void Liste::einfuegen( Daten* d ) {
   anfang->einfuegen(d);
}
```

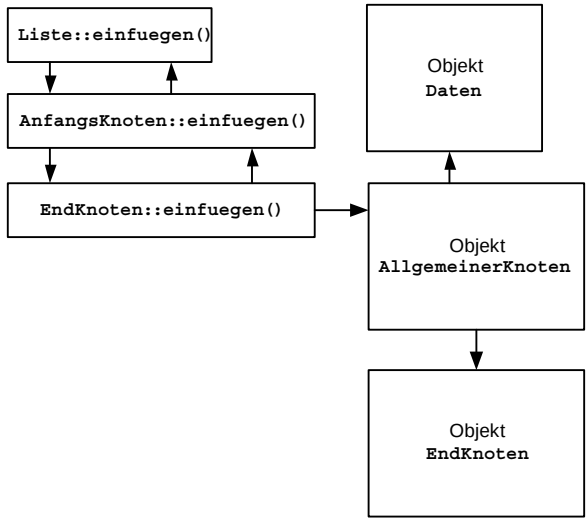

Abbildung 4.16 ... bis die Adresse verworfen wird

Somit sieht diese verkettete Liste nach dem Einfügen eines Objekts der Klasse `Daten` wie folgt aus:

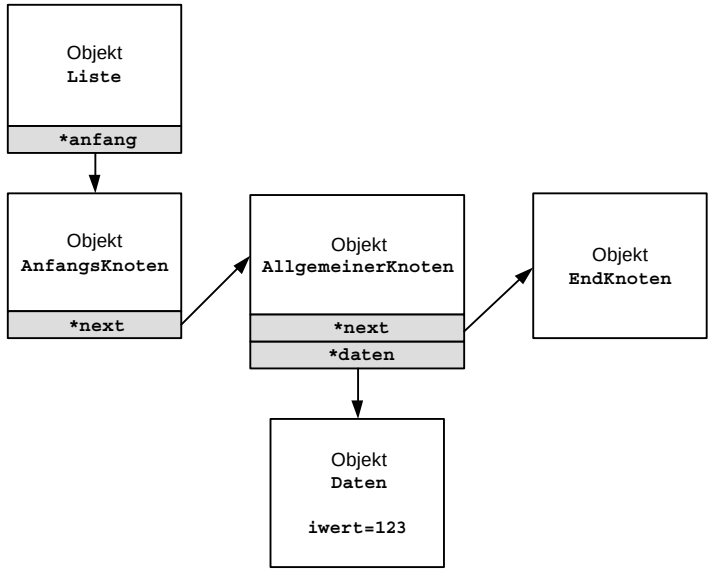

Abbildung 4.17 Der erste eingefügte Knoten der verketteten Liste

Nachdem der erste Knoten eingefügt wurde, kann der nächste Knoten hinzukommen. Da hierbei die Ausführung wieder ein wenig anders ist, soll der Vorgang nochmals an einem zweiten Objekt vom Typ `Daten` demonstriert werden:

```
Objekt [EndKnoten] erzeugt
Objekt [AnfangsKnoten] erzeugt
Wert eingeben (0=Ende): 123
Objekt [Daten] erzeugt
Objekt [AllgemeinerKnoten] erzeugt
Wert eingeben (0=Ende): 12
Objekt [Daten] erzeugt
Objekt [AllgemeinerKnoten] erzeugt
```

Jetzt soll das Objekt `Daten` mit der `iwert`-Eigenschaft `12` in die verkettete Liste eingefügt werden. Hierbei finden Sie zunächst wieder die folgende bekannte Ausführung vor:

```
Liste elemente;
Daten* daten;
...
daten = new Daten(12);
elemente.einfuegen(daten);
```

Es wird wieder die Methode `Liste::einfuegen()` mit dem `Daten`-Objekt aufgerufen (siehe Abbildung 4.18):

```
// class Liste
AnfangsKnoten* anfang;
...
   void Liste::einfuegen( Daten* d ) {
      anfang->einfuegen(d);
   }
```

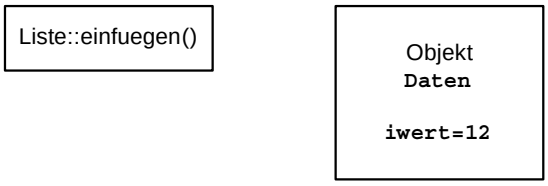

Abbildung 4.18 »Daten«-Objekt in die »Liste« einfügen

Die Liste delegiert diese Arbeit wieder weiter an `AnfangsKnoten::einfuegen()`. Die Methode übergibt das neue `Daten`-Objekt an den Knoten, auf den `next` momentan zeigt:

```
Knoten* AnfangsKnoten::einfuegen( Daten* d ) {
   // Am Anfang kommen keine Daten rein, daher an den
   // nächsten Knoten weiterreichen
   next = next->einfuegen(d);
   return this;
}
```

Der `next`-Zeiger vom Typ `Knoten` verweist im Augenblick auf den Knoten mit dem `Daten`-Objekt (`iwert=123`).

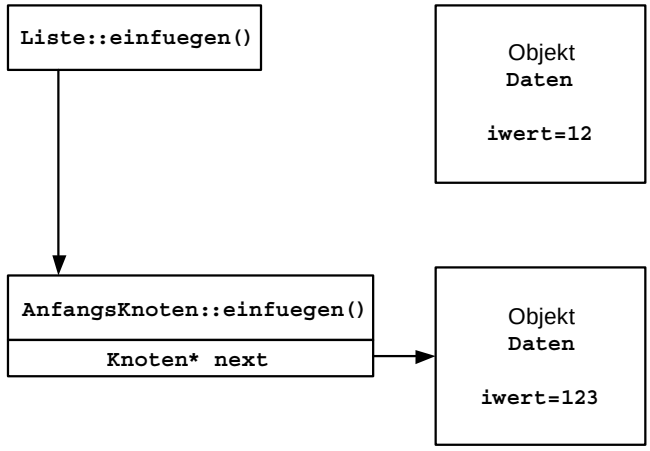

Abbildung 4.19 next-Zeiger »Knoten« verweist auf Daten.

Darauf folgt gleich der nächste Methodenaufruf `AllgemeinerKnoten::einfuegen()`:

```
class AllgemeinerKnoten : public Knoten {
private:
   Daten* daten;
   Knoten* next;
   ...
};

// Die wichtigsten Methoden in diesem Programm
Knoten* AllgemeinerKnoten::einfuegen( Daten* d ) {
   // Wir sortieren aufwärts - kleiner Wert vor großem Wert
   int ret = daten->vergleichen( *d );
   switch( ret ) {
      case 0:     // ... Gleich
      case 1: {   // ... Größer
         // Neue Daten vor den aktuellen einordnen
         AllgemeinerKnoten* dKnoten =
```

```
            new AllgemeinerKnoten( d, this );
         return dKnoten;
      }
      case -1:    //... Kleiner
         // größer als das aktuelle Element -
         // weiter zum nächsten Knoten
         next = next->einfuegen( d );
         return this;
   }
   return this;
}
```

Dabei wird das `Daten`-Objekt – hier mit dem Zeiger `daten` (`iwert=123`) – mit der Methode `Daten::vergleichen()` aufgerufen. Als Argument übergeben Sie dieser Methode die Daten des neuen Objekts (`iwert=12`):

```
// Zum Vergleichen der Eigenschaften zweier Objekte
int Daten::vergleichen( const Daten& d ) {
   if( iwert > d.iwert ) { return 1; };    // ...Größer
   if( iwert < d.iwert ) { return -1;  };  // ...Kleiner
   return 0; // Gleich
}
```

Da das neue `Daten`-Objekt (`d.iwert`) den Wert `12` hat und das bereits in der Liste vorhandene `123` (`iwert`), gibt `Daten::vergleichen()` den Wert `1` zurück – also größer –, woraufhin die `switch`-Anweisung von `AllgemeinerKnoten::einfuegen()` in den folgenden Abschnitt verzweigt:

```
Knoten* AllgemeinerKnoten::einfuegen( Daten* d ) {
   // Wir sortieren aufwärts - kleiner Wert vor großem Wert
   int ret = daten->vergleichen( *d );
   switch( ret ) {
      case 0:     // ... Gleich
      case 1: {   // ... Größer
         // Neue Daten vor den aktuellen einordnen
         AllgemeinerKnoten* dKnoten =
            new AllgemeinerKnoten( d, this );
         return dKnoten;
      }
      ...
}
```

Jetzt wird für das neue `Daten`-Objekt wieder ein `AllgemeinerKnoten` erzeugt. Dieser Knoten (`dKnoten`) verweist jetzt auf das aktuelle `AllgemeinerKnoten`-Objekt und gibt diese Adresse an den Aufrufer `AnfangsKnoten::einfuegen()` zurück.

```
Knoten* AnfangsKnoten::einfuegen( Daten* d ) {
   // Am Anfang kommen keine Daten rein, daher an den
   // nächsten Knoten weiterreichen
   next = next->einfuegen(d);
   return this;
}
```

Jetzt haben Sie den neuen Knoten zur Liste hinzugefügt, wodurch sich folgender »bildlicher« Zustand in der Liste ergibt:

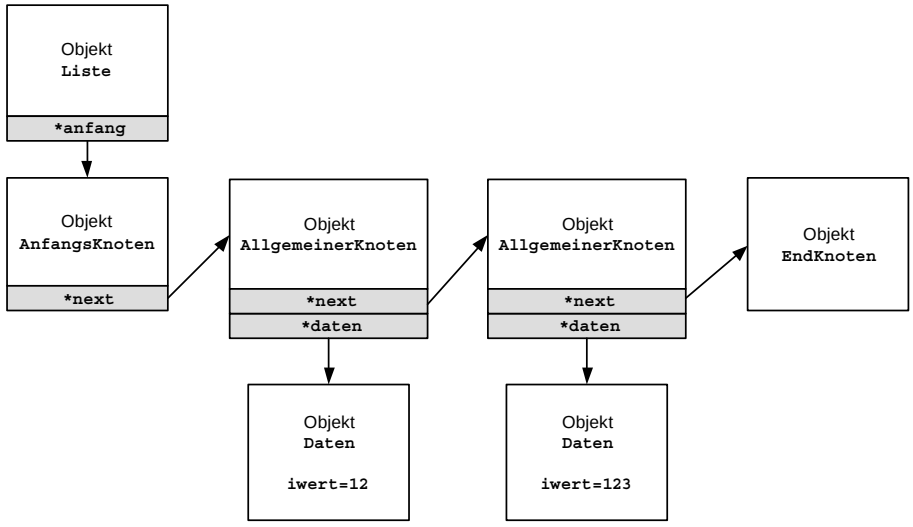

Abbildung 4.20 Neues Objekt hinzugefügt

Ein Durchgang fehlt uns noch, z. B. wenn ein neues Objekt dazwischen eingefügt werden soll (hier zwischen `iwert=12` und `iwert=123`), zum Beispiel der Wert `50`. Der Vorgang ist wie der eben beschriebene mit dem Wert `12`, nur dass in der Methode `AllgemeinerKnoten::einfuegen()` die Methode `Daten::vergleichen()` beim ersten Durchlauf den Wert `–1` zurückgibt, weil das »aktuelle« Objekt (`12`) kleiner ist als das neue Objekt (`50`). Dadurch verzweigt die `switch`-Fallunterscheidung zu `–1`, anstatt wie im Durchlauf zuvor einen neuen `AllgemeinenKnoten` zu erzeugen:

```
// Die wichtigsten Methoden in diesem Programm
Knoten* AllgemeinerKnoten::einfuegen( Daten* d ) {
   // Wir sortieren aufwärts - kleiner Wert vor großem Wert
   int ret = daten->vergleichen( *d );
   switch( ret ) {
      case 0:     // ... Gleich
      case 1: {   // ... Größer
```

```
            // Neue Daten vor den aktuellen einordnen
            AllgemeinerKnoten* dKnoten =
               new AllgemeinerKnoten( d, this );
            return dKnoten;
         }
         case -1:    //... Kleiner
            // größer als das aktuelle Element -
            // weiter zum nächsten Knoten
            next = next->einfuegen( d );
            return this;
      }
      return this;
}
```

Hier werden die neuen Daten als Argument einfach erneut mit der Methode `AllgemeinerKnoten` (Rekursion) mit dem Objekt, auf das der Zeiger `next` hier verweist, aufgerufen. Dies wäre ein weiterer `AllgemeinerKnoten`, der auf ein `Daten`-Objekt mit dem Wert `123` verweist. Ein weiterer Aufruf von `Daten::vergleichen()` steht bevor. Diesmal ist aber das »aktuelle« Objekt (`123`) größer als das neue Objekt (`50`), weshalb wieder `1` zurückgegeben wird. Somit wird jetzt ein neues Objekt der Klasse `AllgemeinerKnoten` erzeugt, mit den `Daten` verknüpft (wenn man das so sagen kann) und an den Aufrufer zurückgegeben:

```
      ...
      case 1: {   // ... Größer
         // Neue Daten vor den aktuellen einordnen
         AllgemeinerKnoten* dKnoten =
            new AllgemeinerKnoten( d, this );
         return dKnoten;
      }
      ...
```

Der Aufrufer war diesmal die Methode `AllgemeinerKnoten::einfuegen()` selbst. Somit wird die Adresse des eingefügten Knotens an das Objekt `AllgemeinerKnoten` weitergereicht, der das `Daten`-Objekt mit dem Wert `12` enthält:

```
     ...
      case -1:    //... Kleiner
         // größer als das aktuelle Element -
         // weiter zum nächsten Knoten
         next = next->einfuegen( d );
         return this;
   }
   ...
```

Das Objekt `AllgemeinerKnoten` mit den `Daten`, die den Wert `12` enthalten, gibt seine eigene Adresse (`this`) an den Aufrufer, hier die Methode `AnfangsKnoten::einfuegen()`, und somit an den `next`-Zeiger zurück. Die Methode `AnfangsKnoten::einfuegen()` wiederum gibt ihre eigene Adresse an den Aufrufer `Liste::einfuegen()` zurück – wo die Adresse »verworfen« wird.

Sicherlich werden Sie sich fragen, warum man hier eine Adresse zurückgibt, nur um diese dann zu verwerfen? Ganz einfach, weil die Methode `einfuegen()` in der Basisklasse `Knoten` deklariert wurde, und zwar mit dem Rückgabewert. Andere Redefinitionen, die von dieser Basisklasse abgeleitet sind, benötigen diesen Rückgabewert an den Aufrufer. Würden Sie den Rückgabewert nur für die Methode `AnfangsKnoten::einfuegen()` ändern, ließe sich das Programm nicht mehr übersetzen.

Das komplette Listing

Zum Abschluss finden Sie hierzu nochmals das komplette Listing. Die Header-Datei *daten.h*, die die Klasse `Daten` enthält, wurde ja bereits komplett abgedruckt. Es fehlt uns also nur noch die Header-Datei *lliste.h*, die die abstrakte Basisklasse `Knoten` mit ihren abgeleiteten Klassen `AnfangsKnoten`, `AllgemeinerKnoten` und `EndKnoten` enthält. Außerdem ist auch die Klasse `Liste` enthalten:

```
// lliste.h
#include <iostream>
#include "daten.h"
#ifndef _LLISTE_H_
#define _LLISTE_H_

class Knoten;
class AnfangsKnoten;
class EndKnoten;
class AllgemeinerKnoten;

// Knoten ist die abstrakte Basisklasse
// Alle abgeleiteten Klassen müssen "einfuegen" und "anzeigen"
   redefinieren
class Knoten {
public:
   Knoten() {}
   virtual ~Knoten() {}
   virtual Knoten* einfuegen( Daten* d ) = 0;
   virtual void anzeigen() = 0;
};
```

```
// ------ Abgeleitete Klasse AllgemeinerKnoten -----------
// Diese Klasse ist für die eigentliche Verwaltung der Daten
// verantwortlich - im Beispiel wird zwar ein Objekt vom Typ
// "Daten" verwendet, aber mit Template (späteres Kapitel)
// können Sie hier noch eins drauf setzen und die Listen-
// Klasse verallgemeinern und somit (fast) unabhängig von den
// Daten machen
class AllgemeinerKnoten : public Knoten {
private:
   Daten* daten;
   Knoten* next;
public:
   // Konstruktor
   AllgemeinerKnoten( Daten* d, Knoten* n ):
                                 daten(d), next(n) {
   // Zu Debug- bzw. Verständniszwecken ggf. entfernen
   cout << "Objekt [AllgemeinerKnoten] erzeugt\n";
   }
   // Destruktor
   ~AllgemeinerKnoten() {
      delete next;
      delete daten;
   }
   // Implizit virtual
   Knoten* einfuegen( Daten* d );
   // Implizit virtual
   void anzeigen();
};

// Die wichtigsten Methoden in diesem Programm
Knoten* AllgemeinerKnoten::einfuegen( Daten* d ) {
   // Wir sortieren aufwärts - kleiner Wert vor großem Wert
   int ret = daten->vergleichen( *d );
   switch( ret ) {
      case 0:
      case 1: {
         // Neue Daten vor den aktuellen einordnen
         AllgemeinerKnoten* dKnoten =
            new AllgemeinerKnoten( d, this );
         return dKnoten;
      }
      case -1:
         // größer als das aktuelle Element - weiter zum
         // nächsten Knoten
         next = next->einfuegen( d );
         return this;
```

```
   }
   return this;
}

void AllgemeinerKnoten::anzeigen() {
   daten->anzeigen();
   next->anzeigen();
}

// ------ Abgeleitete Klasse EndKnoten -------------------
// Der EndKnoten dient im Grunde nur als Endpunkt der Liste
class EndKnoten : public Knoten {
public:
   EndKnoten() {
   // Zu Debug- bzw. Verständniszwecken ggf. entfernen
   cout << "Objekt [EndKnoten] erzeugt\n";
   }
   ~EndKnoten() {}
   // Implizit virtual
   Knoten* einfuegen( Daten* d );
   // Implizit virtual
   void anzeigen() { };
};

// Daten werden immer vor dem Ende eingefügt
Knoten* EndKnoten::einfuegen( Daten* d ) {
   AllgemeinerKnoten* daten =
      new AllgemeinerKnoten(d, this);
   return daten;
}

// ------ Abgeleitete Klasse AnfangsKnoten --------------
// Knoten, der "nur" immer auf das erste Element der
// Liste zeigt
class AnfangsKnoten : public Knoten {
private:
   // ... zeigt immer auf das erste Element
   Knoten* next;
public:
   // Konstruktor
   AnfangsKnoten() {
      // ... gleich auch einen Endknoten erzeugen
      next = new EndKnoten;
      // Zu Debug- bzw. Verständniszwecken ggf. entfernen
      cout << "Objekt [AnfangsKnoten] erzeugt\n";
   }
```

```
   ~AnfangsKnoten() {}
   // Implizit virtual
   Knoten* einfuegen( Daten* d );
   // Implizit virtual
   void anzeigen();
};

Knoten* AnfangsKnoten::einfuegen( Daten* d ) {
   // Am Anfang kommen keine Daten rein, daher an den
   // nächsten Knoten weiterreichen
   next = next->einfuegen(d);
   return this;
}

void AnfangsKnoten::anzeigen() {
   next->anzeigen();
}

// ------ Unabhängige Klasse Liste -------------------
class Liste {
private:
   AnfangsKnoten *anfang;
public:
   // Bei Anlegen gleich ein Objekt "Anfangsknoten" erzeugen
   // Der Konstruktor von "AnfangsKnoten" erzeugt wiederum
   // ein Objekt "EndKnoten", auf das dieser gleich zeigt.
   Liste() { anfang = new AnfangsKnoten; }
   ~Liste() { delete anfang; }
   void einfuegen( Daten* d ) {
      anfang->einfuegen(d);
   }
   void alles_anzeigen() {
      anfang->anzeigen();
   }
};
#endif
```

Jetzt fehlt nur noch das Hauptprogramm:

```
// main.cpp
#include "daten.h"
#include "lliste.h"

int main( void ) {
   Liste elemente;
   Daten* daten;
```

```
   int iwert;
   for(;;) {
      cout << "Wert eingeben (0=Ende): ";
      // Falsche Eingabe oder 0
      if( (!(cin >> iwert)) || iwert ==0 )
         break; // Ende
      daten = new Daten(iwert);
      elemente.einfuegen(daten);
   }
   elemente.alles_anzeigen();
   return 0;
}
```

Das Programm bei der Ausführung:

```
Objekt [EndKnoten] erzeugt
Objekt [AnfangsKnoten] erzeugt
Wert eingeben (0=Ende): 10
Objekt [Daten] erzeugt
Objekt [AllgemeinerKnoten] erzeugt
Wert eingeben (0=Ende): 5
Objekt [Daten] erzeugt
Objekt [AllgemeinerKnoten] erzeugt
Wert eingeben (0=Ende): 8
Objekt [Daten] erzeugt
Objekt [AllgemeinerKnoten] erzeugt
Wert eingeben (0=Ende): 11
Objekt [Daten] erzeugt
Objekt [AllgemeinerKnoten] erzeugt
Wert eingeben (0=Ende): 0
5
8
10
11
```

Zusammenfassung

Dem ein oder anderen dürfte das Beispiel mit den verketteten Listen jetzt eventuell recht viel abverlangt haben. Wer vielleicht verkettete Listen in einer prozeduralen Sprache wie C geschrieben hat, der wird beim OOP-Ansatz gestaunt haben und auch den Vorteil von OOP erkennen.

Anstatt hier Funktion für Funktion aufzurufen und eine Überprüfung nach der anderen vorzunehmen, wie dies in C üblich ist, wird die Verantwortung auf mehrere Klassen aufgeteilt. Jede Klasse hat ihren Verantwortungsbereich, man verwendet hier einfach viele kleine Zahnräder statt eines großen.

Ebenso werden die Daten von der eigentlichen Arbeit der Liste getrennt. Wenn es um die Templates geht, können Sie eine solche Liste sogar unabhängig von den Daten erstellen; das heißt, die Liste können Sie immer wieder verwenden, egal wie die Daten aussehen. Bisher müssen Sie noch immer ein paar Anpassungen vornehmen, sofern Sie andere Daten als die der Klasse `Daten` verwenden wollen.

4.9 Mehrfachvererbung

Bisher haben Sie entweder Klassen neu erstellt, oder die Klassen wurden von der Basisklasse abgeleitet. Es ist aber in C++ auch möglich, eine neue Klasse von mehreren Basisklassen abzuleiten – die Mehrfachvererbung. Damit wird aus mehreren bereits existierenden Klassen eine neue Klasse gebildet.

Die auf diese Weise neu definierte Klasse kann alle Eigenschaften und Methoden der bereits existierenden Klassen übernehmen. Eine mehrfach abgeleitete Klasse lässt sich recht einfach realisieren. Nehmen wir an, Sie haben eine Klasse `Brot` und eine Klasse `Wurst`, dann können Sie diese beiden Klassen zum Beispiel in einer neuen Klasse `Wurstbrot` ableiten. Die Syntax ist der einfachen Ableitung recht ähnlich:

```
class Wurstbrot : public Brot, public Wurst {
   // Weitere Eigenschaften und Methoden
};
```

In diesem Fall besitzt ein Objekt vom Typ `Wurstbrot` die Datenelemente der Klassen `Wurst` und `Brot`.

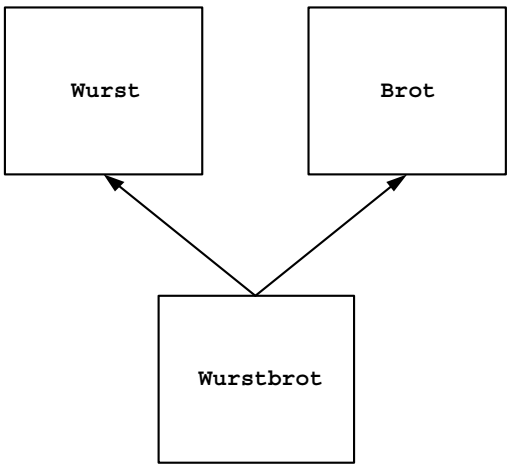

Abbildung 4.21 Mehrfachvererbung

Was vererbt wird, hängt auch hier wieder davon ab, ob Sie `public`, `protected` oder `private` verwenden. Das Prinzip bleibt so, wie Sie es bereits bei den einfachen abgeleiteten Klassen kennengelernt haben. Geben Sie keines dieser drei Schlüsselwörter an, wird automatisch `private` dafür verwendet. In der Deklaration von `Wurstbrot` wurde zweimal `public` verwendet, wodurch in dieser Klasse alle Eigenschaften und Methoden von `Brot` und `Wurst` in der neuen Klasse `Wurstbrot` zur Verfügung stehen, die ebenfalls eine öffentliche Schnittstelle anbietet.

Hier das Beispiel der Klasse `Wurstbrot` in der Praxis:

```
// mehrfachvererb1.cpp
#include <iostream>
#include <cstring>
using namespace std;

class Brot {
protected:
   char brot[100];
   unsigned int gramm;
public:
   void Initialisiere(const char* b="", unsigned int g=0) {
      strncpy( brot, b, 100 );
      brot[100-1] = 0;
      gramm = g;
   };
   unsigned int get_gramm() const { return gramm; };
   const char* get_brot() const { return brot; };
};

class Wurst {
protected:
   char wurst[100];
   unsigned int scheiben;
public:
   void Initialisiere(const char* w="", unsigned int s=0) {
      strncpy( wurst, w, 100 );
      wurst[100-1] = 0;
      scheiben = s;
   };
   unsigned int get_scheiben() const { return scheiben; };
   const char* get_wurst() const { return wurst; };
};
```

```
// Mehrfachvererbung
class Wurstbrot : public Brot, public Wurst {
public:
   void InitAlle( const char* b, unsigned int g,
                  const char* w, unsigned int s ) {
      strncpy( brot, b, 100 );
      gramm = g;
      strncpy( wurst, w, 100 );
      scheiben = s;
   }
   void get_All() const {
      cout << "Gramm    : " << get_gramm() << '\n';
      cout << "Brot     : " << get_brot() << '\n';
      cout << "Scheiben : " << get_scheiben() << '\n';
      cout << "Wurst    : " << get_wurst() << '\n';
   }
};

int main() {
   Wurstbrot wbrot;
   wbrot.InitAlle("Vollkornbrot", 100, "Salami", 3);
   cout << "Gramm    : " << wbrot.get_gramm() << '\n';
   cout << "Brot     : " << wbrot.get_brot() << '\n';
   cout << "Scheiben : " << wbrot.get_scheiben() << '\n';
   cout << "Wurst    : " << wbrot.get_wurst() << '\n';
   cout << endl;
   // ... oder alle auf einmal
   wbrot.get_All();
   return 0;
}
```

Das Programm bei der Ausführung:

```
Gramm    : 100
Brot     : Vollkornbrot
Scheiben : 3
Wurst    : Salami

Gramm    : 100
Brot     : Vollkornbrot
Scheiben : 3
Wurst    : Salami
```

In der Praxis gestaltet sich die Mehrfachvererbung recht einfach. Gewöhnlich entstehen nur Probleme, wenn zwei Basisklassen Elemente mit gleichem Namen

haben. Da der Zugriff auf solche Elemente nicht mehr eindeutig vom Compiler aufgelöst werden kann, müssen Sie hierzu den Scope-Operator verwenden. Wenn Sie beispielsweise in der Klasse `Wurst` die Methode `get_scheiben()` umändern in `get_gramm()` und die Eigenschaft `scheiben` in `gramm`, dann haben Sie je eine Eigenschaft und eine Methode, die mit gleichen Namen in der Klasse `Brot` vorkommen. Im Beispiel müssen Sie daher allen Bezeichnern, die hier nicht eindeutig aufgelöst werden können, die Klasse mit dem Scope-Operator voranstellen:

```
// mehrfachvererb2.cpp
#include <iostream>
#include <cstring>
using namespace std;

class Brot {
protected:
   char brot[100];
   unsigned int gramm;
public:
   void Initialisiere(const char* b="", unsigned int g=0) {
      strncpy( brot, b, 100 );
      brot[100-1] = 0;
      gramm = g;
   };
   unsigned int get_gramm() const { return gramm; };
   const char* get_brot() const { return brot; };
};

class Wurst {
protected:
   char wurst[100];
   unsigned int gramm;
public:
   void Initialisiere(const char* w="", unsigned int g=0) {
      strncpy( wurst, w, 100 );
      wurst[100-1] = 0;
      gramm = g;
   };
   unsigned int get_gramm() const { return gramm; };
   const char* get_wurst() const { return wurst; };
};

// Mehrfachvererbung
class Wurstbrot : public Brot, public Wurst {
```

```
public:
   void InitAlle( const char* b, unsigned int g1,
                  const char* w, unsigned int g2 ) {

      strncpy( brot, b, 100 );
      Brot::gramm = g1;
      strncpy( wurst, w, 100 );
      Wurst::gramm = g2;
   }
   void get_All() const {
      cout << "Gramm   : " << Brot::get_gramm() << '\n';
      cout << "Brot    : " << get_brot() << endl;
      cout << "Gramm   : " << Wurst::get_gramm() << '\n';
      cout << "Wurst   : " << get_wurst() << '\n';
   }
};

int main() {
   Wurstbrot wbrot;
   wbrot.InitAlle("Vollkornbrot", 100, "Salami", 3);
   cout << "Gramm   : " << wbrot.Brot::get_gramm() << '\n';
   cout << "Brot    : " << wbrot.get_brot() << '\n';
   cout << "Gramm   : " << wbrot.Wurst::get_gramm() << '\n';
   cout << "Wurst   : " << wbrot.get_wurst() << '\n';
   cout << '\n';
   // ... oder alle auf einmal
   wbrot.get_All();
   return 0;
}
```

4.9.1 Indirekte Basisklassen erben

Die Mehrfachvererbung lässt sich noch »tiefer« ausführen. So ist es möglich, eine Klasse von verschiedenen Klassen abzuleiten, die dieselbe Basisklasse besitzen – eine mehrfache indirekte Basisklasse.

Hierzu kann die neue Klasse `Wurstbrot` von der Klasse `Brot` und `Wurst` abgeleitet werden (wie bisher). Diese beiden Klassen wiederum sind von der Klasse `Supermarkt` abgeleitet, wo der entsprechende Gegenstand gekauft wurde (siehe Abbildung 4.22).

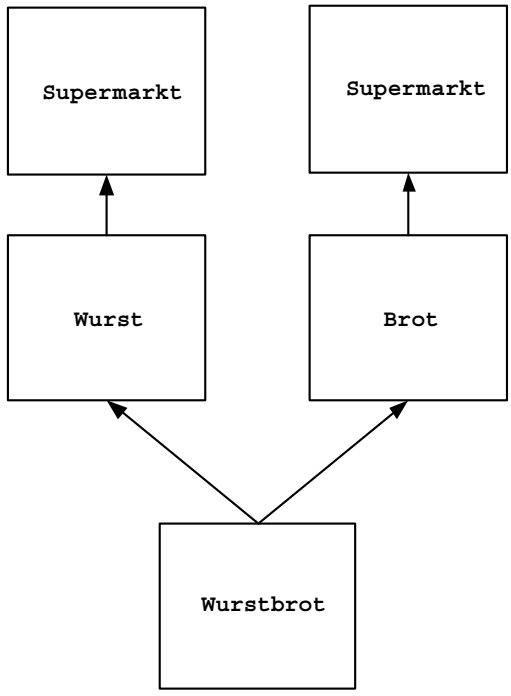

Abbildung 4.22 Mehrfache indirekte Basisklasse

Somit besitzt jetzt ein Objekt vom Typ `Wurstbrot` die Elemente der Klasse `Supermarkt` zweimal. Natürlich bedeutet dies wiederum, dass der Aufruf von Eigenschaften und Methoden der Klasse `Supermarkt` zweideutig ist, so dass ein Aufruf wie

```
Objekt.Methode_von_Supermarkt();          // Fehler !!!
```

zwangsläufig zu einem Fehler des Compilers führt, weil dieser wieder nicht weiß, ob die abgeleitete Methode von `Brot` oder `Wurst` gemeint ist. Dieses Problem wird wieder mit dem Scope-Operator gelöst:

```
Objekt.Brot::Methode_von_Supermarkt();
Objekt.Wurst::Methode_von_Supermarkt();
```

Hierzu wieder ein komplettes Listing:

```
// mehrfachvererb3.cpp
#include <iostream>
#include <cstring>
using namespace std;
```

```
class Supermarkt {
protected:
   char markt[100];
public:
   Supermarkt(const char* m="") {
      strncpy( markt, m, 100 );
      markt[100-1] = 0;
   }
   ~Supermarkt() {}
   const char* get_markt() const { return markt; }
};

class Brot : public Supermarkt {
protected:
   char brot[100];
   unsigned int gramm;
public:
   Brot( const char* b="", unsigned int g=0,
         const char* m="" ): Supermarkt(m) {
      strncpy( brot, b, 100 );
      brot[100-1] = 0;
      gramm = g;
   };
   ~Brot() {}
   unsigned int get_gramm() const { return gramm; };
   const char* get_brot() const { return brot; };
};

class Wurst : public Supermarkt {
protected:
   char wurst[100];
   unsigned int gramm;
public:
   Wurst( const char* w="", unsigned int g=0,
          const char* m=""): Supermarkt(m) {
      strncpy( wurst, w, 100 );
      wurst[100-1] = 0;
      gramm = g;
   };
   ~Wurst() {}
   unsigned int get_gramm() const { return gramm; };
   const char* get_wurst() const { return wurst; };
};

// Mehrfachvererbung
```

```
class Wurstbrot : public Brot, public Wurst {
public:
   Wurstbrot(const char* b,unsigned int g1,const char* s1,
             const char* w,unsigned int g2,const char* s2 ):
             Brot(b, g1, s1), Wurst(w, g2, s2) {  }
   ~Wurstbrot() {}
   void get_All() const {
      cout << "Gramm       : " << Brot::get_gramm() << '\n';
      cout << "Brot        : " << get_brot() << '\n';
      cout << "Gekauft bei : " << Brot::get_markt() << '\n';
      cout << "Gramm       : " << Wurst::get_gramm()
           << '\n';
      cout << "Wurst       : " << get_wurst() << '\n';
      cout << "Gekauft bei : " << Wurst::get_markt()
           << '\n';
   }
};

int main() {
   Wurstbrot wbrot("Vollkornbrot", 100, "Meier",
                   "Salami", 3, "Eberhardt");

   cout << "Gramm       : " << wbrot.Brot::get_gramm()
        << '\n';
   cout << "Brot        : " << wbrot.get_brot() << '\n';
   cout << "Gekauft bei : " << wbrot.Brot::get_markt()
        << '\n';
   cout << "Gramm       : " << wbrot.Wurst::get_gramm()
        << '\n';
   cout << "Wurst       : " << wbrot.get_wurst() << '\n';
   cout << "Gekauft bei : " << wbrot.Wurst::get_markt()
        << '\n';
   cout << '\n';
   // ... oder alle auf einmal
   wbrot.get_All();
   return 0;
}
```

Das Programm bei der Ausführung:

```
Gramm       : 100
Brot        : Vollkornbrot
Gekauft bei : Meier
Gramm       : 20
Wurst       : Salami
Gekauft bei : Eberhardt
```

```
Gramm       : 100
Brot        : Vollkornbrot
Gekauft bei : Meier
Gramm       : 20
Wurst       : Salami
Gekauft bei : Eberhardt
```

4.9.2 Virtuelle indirekte Basisklassen erben

Manchmal ist es allerdings nicht wünschenswert, dass eine Klasse eine indirekte Basisklasse mehrfach erhält, wie dies im Abschnitt zuvor gezeigt wurde. Warum sollten Sie für die Klassen `Brot` und `Wurst` jeweils einen eigenen `Supermarkt` angeben, wenn Sie beides im selben Geschäft kaufen können. In der Praxis ist es meistens sinnvoller, wenn die indirekten Basisklassen nur einmal in der mehrfach abgeleiteten Klasse vorkommen. Dies wird mit Hilfe von virtuellen Basisklassen erreicht.

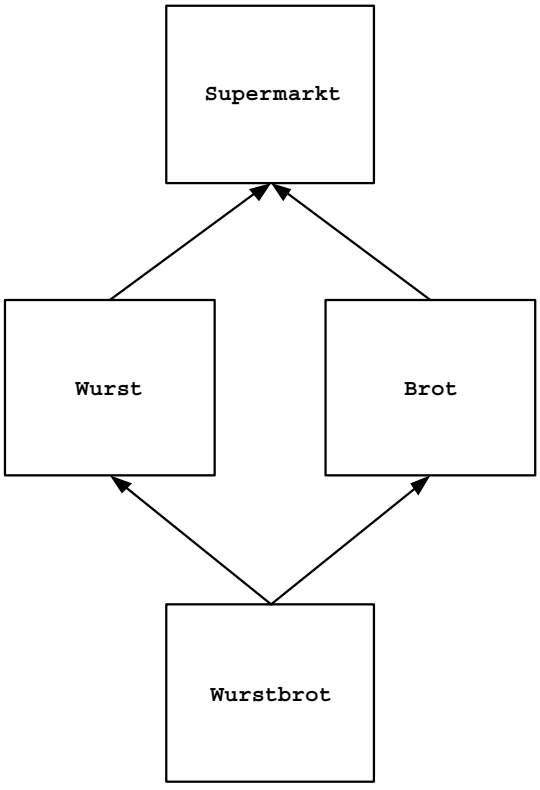

Abbildung 4.23 Indirekte virtuelle Basisklasse »Supermarkt«

Hierzu ist im Grunde (gegenüber der Version zuvor) nicht viel zu verändern. Bei der Deklaration der virtuellen Basisklassen muss nur das Schlüsselwort `virtual` vor den Namen der Basisklasse gesetzt werden:

```
class Brot : public virtual Supermarkt {
   // ...
};

class Wurst : public virtual Supermarkt {
   // ...
};
```

Jetzt besitzt eine von `Wurst` oder `Brot` abgeleitete Klasse die Klasse `Supermarkt` als virtuelle Basisklasse.

Beachten Sie allerdings, dass sich die Virtualität einer Klasse erst bei der Mehrfachvererbung auswirkt bzw. Sinn macht. Objekte von `Wurst` oder `Brot` haben nach wie vor ihren eigenen `Supermarkt`. Ein Objekt vom Typ `Wurstbrot` enthält diese virtuelle Basisklasse allerdings nur noch einmal. Somit können die `public`-Eigenschaften und Methoden jetzt ohne den Scope-Operator verwendet werden, da die von `Supermarkt` geerbten Elemente nur noch einmal im Speicher vorhanden sind, es gibt jetzt nur noch einen `Supermarkt`.

Hierzu das Listing, das die indirekte virtuelle Basisklasse im Einsatz zeigt:

```
// mehrfachvererb4.cpp
#include <iostream>
#include <cstring>
using namespace std;

class Supermarkt {
protected:
   char markt[100];
public:
   Supermarkt(const char* m="") {
      strncpy( markt, m, 100 );
      markt[100-1] = 0;
   }
   ~Supermarkt() {}
   const char* get_markt() const { return markt; }
};

class Brot : public virtual Supermarkt {
protected:
   char brot[100];
   unsigned int gramm;
```

```
public:
   Brot( const char* b="", unsigned int g=0,
         const char* m="" ): Supermarkt(m) {
      strncpy( brot, b, 100 );
      brot[100-1] = 0;
      gramm = g;
   };
   ~Brot() {}
   unsigned int get_gramm() const { return gramm; };
   const char* get_brot() const { return brot; };
};


class Wurst : public virtual Supermarkt {
protected:
   char wurst[100];
   unsigned int gramm;
public:
   Wurst( const char* w="", unsigned int g=0,
          const char* m=""): Supermarkt(m) {
      strncpy( wurst, w, 100 );
      wurst[100-1] = 0;
      gramm = g;
   };
   ~Wurst() {}
   unsigned int get_gramm() const { return gramm; };
   const char* get_wurst() const { return wurst; };
};

// Mehrfachvererbung
class Wurstbrot : public Brot, public Wurst {
public:
   Wurstbrot( const char* b, unsigned int g1,
              const char* w, unsigned int g2,
              const char* s ):
              Brot(b, g1), Wurst(w, g2), Supermarkt(s) {  }
   ~Wurstbrot() {}
   void get_All() const {
      cout << "Gramm       : " << Brot::get_gramm() << '\n';
      cout << "Brot        : " << get_brot() << '\n';
      cout << "Gramm       : " << Wurst::get_gramm()
           << '\n';
      cout << "Wurst       : " << get_wurst() << '\n';
      cout << "Gekauft bei : " << get_markt() << '\n';
   }
```

```
};

int main() {
   Wurstbrot wbrot("Vollkornbrot", 100,
                   "Salami", 20, "Eberhardt");

   cout << "Gramm       : " << wbrot.Brot::get_gramm()
        << '\n';
   cout << "Brot        : " << wbrot.get_brot() << '\n';
   cout << "Gramm       : " << wbrot.Wurst::get_gramm()
        << '\n';
   cout << "Wurst       : " << wbrot.get_wurst() << '\n';
   cout << "Gekauft bei : " << wbrot.get_markt() << '\n';
   cout << '\n';

   // ... oder alle auf einmal
   wbrot.get_All();
   return 0;
}
```

Das Programm bei der Ausführung:

```
Gramm       : 100
Brot        : Vollkornbrot
Gramm       : 20
Wurst       : Salami
Gekauft bei : Eberhardt

Gramm       : 100
Brot        : Vollkornbrot
Gramm       : 20
Wurst       : Salami
Gekauft bei : Eberhardt
```

Reihenfolge bei der Erzeugung der Objekte

Die Erzeugung von Objekten bei der Mehrfachvererbung geschieht (gemäß Abbildung 4.22 bzw. 4.23) von oben nach unten. Beim Erzeugen eines Objekts werden also zunächst die Teilobjekte erzeugt, die von der Basisklasse geerbt wurden. In Bezug auf die Konstruktoren, die immer als Erstes aufgerufen werden, gilt:

- Erst werden die Konstruktoren der (indirekten) virtuellen Basisklassen ausgeführt (von oben nach unten).

- Als Nächstes werden die Konstruktoren der nicht virtuellen direkten Basisklassen aufgerufen.
- Am Ende wird dann der Konstruktor der »eigenen« Klasse aufgerufen.

Bezogen auf unser Beispiel *mehrfachvererb4.cpp*, wird beim Erzeugen eines `Wurstbrot`-Objekts zunächst der Konstruktor der indirekten virtuellen Basisklasse `Supermarkt` aufgerufen. Anschließend werden die beiden Konstruktoren der direkten virtuellen Basisklassen `Wurst` und `Brot` aufgerufen. Am Ende erfolgt der Aufruf des Konstruktors der eigenen Klasse `Wurstbrot`.

Mit Templates haben Sie eine Technik, mit der Sie Klassen und Funktionen unabhängig vom konkreten Datentyp erzeugen können. In diesem Kapitel erfahren Sie alles über das Template-Konzept von C++.

5 Templates und STL

5.1 Funktions-Templates

Da C++ eine typgebundene Sprache ist, kommt es häufig vor, dass man dieselbe Funktion mehrmals, nur mit unterschiedlichem Typ, implementiert. Wenn Sie zum Beispiel Algorithmen für das Sortieren oder Suchen von Arrays implementieren und hierbei eine Version für `long` und eine für `float` anbieten wollen, müssen Sie zwei Versionen der Funktion anbieten oder aber Funktions-Templates verwenden.

Das Prinzip soll anhand des folgenden Beispiels demonstriert werden:

```
// func_template1.cpp
#include <iostream>
using namespace std;

long BigNum( long n1, long n2 );
float BigNum (float n1, float n2);
void Swap( long& n1, long& n2);
void Swap( float& n1, float& n2);

int main() {
   long lnum1 = 100, lnum2 = 111;
   float fnum1 = 100.1, fnum2 = 111.1;

   cout << "Größerer Wert (long) : "
        << BigNum( lnum1, lnum2) << "\n";
   cout << "Größerer Wert (float): "
        << BigNum( fnum1, fnum2) << "\n\n";

   cout << "lnum1: " << lnum1 << " lnum2: "
        << lnum2 << "\n";
   cout << "Tausche Werte (long)\n";
```

```
   Swap( lnum1, lnum2 );
   cout << "lnum1: " << lnum1 << " lnum2: "
        << lnum2 << "\n";

   cout << "fnum1: " << fnum1 << " fnum2: "
        << fnum2 << "\n";
   cout << "Tausche Werte (float)\n";
   Swap( fnum1, fnum2 );
   cout << "fnum1: " << fnum1 << " fnum2: "
        << fnum2 << "\n";
   return 0;
}

long BigNum( long n1, long n2 ) {
   if( n1 > n2 ) { return n1; }
   else if( n1 < n2 ) { return n2; }
}

float BigNum( float n1, float n2 ) {
   if( n1 > n2 ) { return n1; }
   else if( n1 < n2 ) { return n2; }
}

void Swap( long& n1, long& n2 ) {
   long tmp = n1;
   n1 = n2;
   n2 = tmp;
}

void Swap( float& n1, float& n2 ) {
   float tmp = n1;
   n1 = n2;
   n2 = tmp;
}
```

Hier haben Sie zwei Versionen der Funktion `BigNum()`, die den größeren von zwei `long`- bzw. `float`-Werten zurückgibt, und der Funktion `Swap()`, die zwei `long`- bzw. `float`-Werte untereinander austauschen. Gewöhnlich werden Sie auch noch eine Version für `short` hinzufügen wollen.

Allerdings haben Sie die Möglichkeit, Funktions-Templates zu verwenden. Funktions-Templates können Sie sich wie eine Schablone vorstellen; sie sind eine Vorlage gleichartiger Funktionen, die sich im Rückgabewert der Funktion, im Datentyp der Parameter und/oder im Datentyp der lokalen Variablen der Funktion unterscheiden. Außerdem muss (im Gegensatz zur Funktions-Überladung) die

aus einem Funktions-Template erzeugte Funktion die gleiche Anzahl an Parametern haben und auch die gleichen Anweisungen enthalten.

Dadurch ergeben sich für den Programmierer folgende Vorteile:

- Durch das Funktions-Template wird der Code kürzer, da dieser nur einmal programmiert werden muss und dann für verschiedene Typen zur Verfügung steht.
- Die Funktionen werden aus dem Template automatisch erzeugt, sobald diese verwendet werden.
- Weniger Code und weniger Funktionen bedeuten auch weniger Aufwand beim Suchen nach Fehlern und beim Testen des Programms. Der Wartungsaufwand wird ebenfalls geringer.

5.1.1 Funktions-Templates definieren

Funktions-Templates werden mit dem Präfix

```
template <class T>
```

eingeleitet. Der Parameter `T` steht hier für den Typnamen, der in der gleich folgenden Definition verwendet wird. Somit sieht die Definition der Funktions-Templates `BigNum()` wie folgt aus:

```
template <class T>
T BigNum( T n1, T n2 ) {
   if( n1 > n2 ) { return n1; }
   else if( n1 < n2 ) { return n2; }
}
```

Verglichen mit der ursprünglichen Definition

```
long BigNum( long n1, long n2 ) {
   if( n1 > n2 ) { return n1; }
   else if( n1 < n2 ) { return n2; }
}

float BigNum( float n1, float n2 ) {
   if( n1 > n2 ) { return n1; }
   else if( n1 < n2 ) { return n2; }
}
```

wurde `T` verwendet statt eines Datentyps beim Rückgabewert und des Parameters. Das Gleiche ist auch bei lokalen Variablen möglich, wie sie bei der Funktion `Swap()` verwendet und benötigt werden. Die Funktions-Templates für `Swap()` sehen demnach wie folgt aus:

```
template <class T>
void Swap( T& n1, T& n2) {
   T tmp = n1;
   n1 = n2;
   n2 = tmp;
}
```

Der Parametername `T` in der Definition der Funktions-Templates wird wie ein normaler Typname verwendet und muss nicht zwangsläufig `T` heißen. Hier können Sie bezeichnerübliche Namen verwenden – in der Praxis findet man aber häufig den Typnamen `T` vor. Hier das komplette Listing *func_template1.cpp* neu, jetzt mit den Funktions-Templates:

```
// func_template2.cpp
#include <iostream>
using namespace std;

template <class T> T BigNum( T n1, T n2 );
template <class T> void Swap( T& n1, T& n2);

int main() {
   long lnum1 = 100, lnum2 = 111;
   float fnum1 = 100.1, fnum2 = 111.1;

   cout << "Größerer Wert (long) : "
        << BigNum( lnum1, lnum2) << "\n";
   cout << "Größerer Wert (float): "
        << BigNum( fnum1, fnum2) << "\n\n";

   cout << "lnum1: " << lnum1 << " lnum2: "
        << lnum2 << "\n";
   cout << "Tausche Werte (long)\n";
   Swap( lnum1, lnum2 );
   cout << "lnum1: " << lnum1 << " lnum2: "
        << lnum2 << "\n";

   cout << "fnum1: " << fnum1 << " fnum2: "
        << fnum2 << "\n";
   cout << "Tausche Werte (float)\n";
   Swap( fnum1, fnum2 );
   cout << "fnum1: " << fnum1 << " fnum2: "
        << fnum2 << "\n";
   return 0;
}
```

```
template <class T>
T BigNum( T n1, T n2 ) {
   if( n1 > n2 ) { return n1; }
   else if( n1 < n2 ) { return n2; }
}

template <class T> void Swap( T& n1, T& n2) {
   T tmp = n1;
   n1 = n2;
   n2 = tmp;
}
```

Mit der Definition eines Funktions-Templates wird noch lange keine korrekte Funktion erzeugt. Der Maschinencode wird erst erzeugt, wenn eine konkrete Funktion für einen bestimmten Typ benötigt wird. Erfolgt zum Beispiel niemals ein Funktionsaufruf `Swap()` für den Datentyp `float`, so wird auch keine solche Funktion als Maschinencode erzeugt. Man sagt, ein Funktions-Template wird instantiiert, also für einen bestimmten Typ generiert, wenn dieses Template zum ersten Mal aufgerufen wird.

Das Ermitteln des entsprechenden Datentyps übernimmt dabei der Compiler:

```
int iwert1 = 100, iwert2 = 200;
Swap( iwert1, iwert2 );    // Werte tauschen
```

Jetzt erzeugt der Compiler eine Funktion aus dem Funktions-Template `Swap()` für den Datentyp `int`. Der Compiler ersetzt im Maschinencode den Parameter `T` durch das Template-Argument `int`. Wird erneut eine Funktion mit den Argumenten `int` benötigt, so wird die bereits generierte Funktion aufgerufen. Wird jetzt die Funktion `Swap()` mit `float`-Argumenten aufgerufen, wird eine weitere Funktion aus dem Funktions-Template für den Datentyp `float` vom Compiler generiert.

Natürlich ist es auch erlaubt, solche Funktions-Templates als `inline` zu deklarieren:

```
template <class T>
inline void Swap( T& n1, T& n2) {
   T tmp = n1;
   n1 = n2;
   n2 = tmp;
}
```

Des Weiteren ist es möglich, dass in einem Funktions-Template ein weiteres Funktions-Template aufgerufen wird – man spricht dabei vom Verschachteln von Funktions-Templates:

```
// func_template3.cpp
#include <iostream>
using namespace std;

template <class T> bool BigNum( T n1, T n2 );
template <class T> void Swap( T& n1, T& n2);

int main() {
   long lnum1 = 111, lnum2 = 11;
   float fnum1 = 11.1, fnum2 = 111.1;

   cout << lnum1 << " " << lnum2 << "\n";
   cout << fnum1 << " " << fnum2 << "\n";

   // Tauschen, falls nötig
   Swap( lnum1, lnum2 );
   Swap( fnum1, fnum2 );

   cout << lnum1 << " " << lnum2 << "\n";
   cout << fnum1 << " " << fnum2 << "\n";
   return 0;
}

template <class T>
bool BigNum( T n1, T n2 ) {
   if( n1 > n2 ) { return true; }
   else { return false; }
}

template <class T> void Swap( T& n1, T& n2) {
   if( BigNum( n1, n2) ) {
      T tmp = n1;
      n1 = n2;
      n2 = tmp;
   }
}
```

Hier werden zwei Werte nur getauscht, wenn der linke Wert größer als der rechte Wert in der Argumentenliste ist.

5.1.2 Typübereinstimmung

Beachten Sie, dass beim Auflösen eines Templates vom Compiler niemals eine automatische Typkonvertierung vorgenommen wird. Das neue Funktions-Template muss immer so generiert werden, dass die Datentypen der Parameter mit

den Typen der Argumente übereinstimmen. Selbst einfachste implizite Typanpassungen von `int` nach `long` führen hier zu einem Compiler-Fehler:

```
int ival = 100;
long lval = 200;
Swap( ival, lval ); // Fehler !!!
```

Hier müsste Folgendes generiert werden:

```
void Swap( int&, long& );
```

Aber bei der Definition des Funktions-Templates wurde mit `T` nur ein Typ angegeben. Hierbei können Sie entweder selbst eine Typanpassung vornehmen, wie beispielsweise

```
Swap( (long) ival, lval);
// ... oder ...
Swap( 100, lval );
```

oder aber Sie erstellen ein Funktions-Template, das verschiedene Parameter aufnehmen kann (siehe Abschnitt 5.1.5, »Verschiedene Parameter«).

5.1.3 Funktions-Templates über mehrere Module

Wollen Sie Funktions-Templates über mehrere Module generieren, sollten Sie die Definition des Templates in eine Header-Datei stellen, damit das Template in allen Modulen zur Verfügung steht, in denen die Header-Datei eingebunden wird.

Der Grund dafür ist, dass die endgültige Funktion (Maschinencode) erst beim Aufruf einer durch das Funktions-Template vorgegebenen Funktion erstellt wird. Und dafür benötigt der Compiler den Code der Funktion.

5.1.4 Spezialisierung von Funktions-Templates

In manchen Fällen wird eine Spezialisierung der Funktions-Templates nötig. Wollen Sie zum Beispiel mit einem Funktions-Template zwei Objekte einer Klasse tauschen, müssen für diese Klasse der Kopierkonstruktor und die Zuweisung vorhanden sein. Hier liegt also die Spezialisierung nicht an dem Funktions-Template, sondern an der Klasse.

Bei dem folgenden Beispiel muss man schon zweimal überlegen, ob es korrekt ist, wenn man Funktions-Templates auch auf C-Strings anwendet:

```
// func_template4.cpp
#include <iostream>
using namespace std;
```

```
template <class T> void Swap( T& n1, T& n2);

int main() {
   long lnum1 = 111, lnum2 = 11;
   const char* str1 = "ABCD";
   const char* str2 = "EFGH";
   cout << lnum1 << " " << lnum2 << "\n";
   Swap( lnum1, lnum2 );
   cout << lnum1 << " " << lnum2 << "\n";
   cout << str1 << " " << str2 << "\n";
   Swap( str1, str2 );
   cout << str1 << " " << str2 << "\n";
   return 0;
}

template <class T> void Swap( T& n1, T& n2) {
      T tmp = n1;
      n1 = n2;
      n2 = tmp;
}
```

Zugegeben, das Beispiel funktioniert noch. Anders sieht es allerdings mit dem folgenden Beispiel aus:

```
// func_template5.cpp
#include <iostream>
using namespace std;

template <class T> T BigObjekt( T n1, T n2 );

int main() {
   long lnum1 = 111, lnum2 = 11;
   const char* str1 = "ABCD";
   const char* str2 = "AAAA";
   cout << BigObjekt(lnum1, lnum2) << "\n";
   cout << BigObjekt(str1, str2) << "\n";
   return 0;
}

template <class T>
T BigObjekt( T n1, T n2 ) {
   if( n1 > n2 ) { return n1; }
   else if( n1 < n2 ) { return n2; }
}
```

Auch wenn es hier den Anschein hat, dass alles richtig abläuft, sollten Sie bedenken, dass lediglich die größere der beiden Adressen zurückgegeben wird, unter denen die Strings gespeichert sind. Hier haben Sie nun die Möglichkeit, eine Spezialisierung einzubauen. Dafür müssen Sie theoretisch nur eine separat definierte Funktion überladen:

```
// func_template6.cpp
#include <iostream>
#include <cstring>
using namespace std;

template <class T> T BigObjekt( T n1, T n2 );
const char* BigObjekt( const char* s1, const char* s2);

int main() {
   long lnum1 = 111, lnum2 = 11;
   const char* str1 = "ABCD";
   const char* str2 = "AAAA";
   cout << BigObjekt(lnum1, lnum2) << "\n";
   cout << BigObjekt(str1, str2) << "\n";
   return 0;
}

template <class T>
T BigObjekt( T n1, T n2 ) {
   if( n1 > n2 ) { return n1; }
   else if( n1 < n2 ) { return n2; }
}

const char* BigObjekt( const char* s1, const char* s2 ) {
   if( strcmp( s1, s2) > 0 ) { return s1; }
   else { return s2; }
}
```

Die Reihenfolge, in der der Compiler nach einer Funktion sucht, garantiert Ihnen, dass die spezialisierte Funktion stets vor dem Funktions-Template verwendet wird, wenn der entsprechende Typ (hier `char*`) deklariert wurde.

Allerdings wirft dies bei umfangreicheren Projekten, die über mehrere Module programmiert wurden, ein Problem auf. Wird nämlich die Spezialisierung in einem anderen Modul als das Funktions-Template definiert, weiß der Compiler nicht, ob eine Deklaration einer Template-Instanz oder eine Spezialisierung vorliegt.

Daher gibt es im neuesten ANSI-Standard eine eigene Syntax dafür, wie eine Spezialisierung zu definieren ist. Natürlich bedeutet dies auch, dass dieser Vorgang noch nicht von allen Compilern unterstützt wird. Eine solche Spezialisierung beginnt mit dem Präfix:

```
template <>
```

Bezogen auf die Funktion `BigObjekt()`, sieht die Spezialisierung dieser Funktion wie folgt aus:

```
// func_template7.cpp
#include <iostream>
#include <cstring>
using namespace std;

template <class T> T BigObjekt( T n1, T n2 );
template <>
const char* BigObjekt( const char* s1, const char* s2);

int main() {
   long lnum1 = 111, lnum2 = 11;
   const char* str1 = "ABCD";
   const char* str2 = "AAAA";
   cout << BigObjekt(lnum1, lnum2) << "\n";
   cout << BigObjekt(str1, str2) << "\n";
   return 0;
}

template <class T>
T BigObjekt( T n1, T n2 ) {
   if( n1 > n2 ) { return n1; }
   else if( n1 < n2 ) { return n2; }
}

template <>
const char* BigObjekt( const char* s1, const char* s2 ) {
   if( strcmp( s1, s2) > 0 ) { return s1; }
   else { return s2; }
}
```

Diese Spezialisierung wird in folgenden Fällen eingesetzt:

- Die übliche Methode über das Funktions-Template liefert kein vernünftiges Ergebnis.
- Im Funktions-Template gibt es Anweisungen, die auf einen bestimmten Typ nicht ausgeführt werden können (beispielsweise `strcmp()` für Zahlen).

5.1.5 Verschiedene Parameter

Funktions-Templates sind nicht nur auf einen formalen Datentyp beschränkt, sondern können auch mit mehreren Typparametern definiert werden:

```
// func_template8.cpp
#include <iostream>
#include <cstring>
using namespace std;

template <class T1, class T2>
void funktion( T1 n1, T2 n2 );

int main() {
   funktion( 100, 111.111 );
   funktion( "Hallo Welt", 123 );
   funktion( 'p', 3.14 );
   funktion( 111.111, 222.222 );
   return 0;
}

template <class T1, class T2>
void funktion( T1 n1, T2 n2 ) {
   cout << n1 << " : " << n2 << "\n";
}
```

Beide Parameter, `T1` und `T2`, werden in der Definition wie normale Typnamen verwendet. Natürlich ist es auch möglich, hierbei zwei gleichwertige Argumente zu verwenden. Zum Beispiel ist ein Aufruf von

```
int ival1 = 100, ival2 = 200;
funktion( ival1, ival2 ) ;
```

kein Fehler, auch wenn es unnötig erscheint, dass hier zwei Typparameter verwendet werden.

Es sollte außerdem nicht unerwähnt bleiben, dass Sie bei Funktions-Templates auch »gewöhnliche« Parameter mit allen üblichen Features verwenden können:

```
template <class T>
void funktion( T n1, long n2 ) {
   // ...
}
```

5.1.6 Explizite Template-Argumente

Die bisherige Ableitung der Funktions-Templates war implizit. Das bedeutet, ein Funktions-Template wurde mit einem bestimmten Typ instantiiert, wenn es zum ersten Mal aufgerufen wurde. Der Compiler ermittelte dann selbst den Typ für den (oder die) Parameter `T` anhand der Funktionsargumente.

Sie haben aber auch zusätzlich die Möglichkeit, das oder die Template-Argument(e) – gemäß ANSI-Standard – explizit anzugeben. Dabei werden die Template-Argumente in spitzen Klammern hinter dem Template-Namen eingesetzt. Hierzu soll nochmals das folgende Funktions-Template verwendet werden:

```
template <class T>
T BigObjekt( T n1, T n2 ) {
   if( n1 > n2 ) { return n1; }
   else if( n1 < n2 ) { return n2; }
}
```

Würden Sie dieses Funktions-Template folgendermaßen aufrufen

```
BigObjekt( 100, 111.111 );      // Fehler -> (long, double)
BigObjekt( 111.111, 222.222 );  // Ok -> (double, double)
BigObjekt('A', 67);             // Fehler -> (char, int)
```

so würde der Compiler das Listing nicht übersetzen, weil zweimal verschiedene Template-Argumente verwendet wurden, was in der Definition nicht vereinbart war. Wollen Sie diesen Aufruf dennoch erzwingen, können Sie explizite Template-Argumente verwenden. Mit

```
BigObjekt<float>( 100, 111.111 );
```

erzwingen Sie zum Beispiel, dass ein Funktions-Template für den Datentyp `float` generiert wird, egal, ob Sie hier einen anderen Datentyp als Argument verwendet haben. Natürlich funktioniert das auch mit mehreren Parametern. Hierzu ein Listing, das die expliziten Argumente für Funktions-Templates in der Praxis demonstriert:

```
// func_template9.cpp
#include <iostream>
#include <cstring>
using namespace std;

template <class T>
T BigObjekt( T n1, T n2 );
template <class T1, class T2>
void funktion( T1 n1, T2 n2 );
```

```
int main() {
   // Funktion für float generieren
   cout << BigObjekt<float>( 100, 111.111 ) << "\n";
   // Funktion für int generieren
   cout << BigObjekt<int>( 111.111, 222.222 ) << "\n";
   // Funktion für char generieren
   cout << BigObjekt<char>('A', 67) << "\n";
   // Funktion für ein int und ein char generieren
   funktion<int, char>('A', 65);
   // Funktion für zwei int generieren
   funktion<int, int>(11.11, 'B');
   return 0;
}

template <class T>
T BigObjekt( T n1, T n2 ) {
   if( n1 > n2 ) { return n1; }
   else if( n1 < n2 ) { return n2; }
}

template <class T1, class T2>
void funktion( T1 n1, T2 n2 ) {
   cout << n1 << " : " << n2 << "\n";
}
```

Das Programm bei der Ausführung:

```
111.111
222
C
65 : A
11 : 66
```

Hinweis [«]

Beachten Sie, dass viele der hier beschriebenen Nutzungsmöglichkeiten der Funktions-Templates erst bei den neueren Compilern (ab ca. 1999) zur Verfügung stehen.

5.2 Klassen-Templates

Die Templates, die Sie im letzten Abschnitt kennengelernt haben, sind nicht nur auf Funktionen beschränkt, sondern können auch mit Klassen verwendet werden. Was zunächst recht suspekt erscheint, wird relativ häufig verwendet, wenn es um die Entwicklung von Klassenbibliotheken geht. Recht populäre Beispiele

von Klassen-Templates dürften wohl die STL-Bibliothek – siehe Abschnitt 5.3, »STL (Standard Template Library)« – und die Stream-Klassen sein, die alle standardmäßig als Template implementiert sind.

Im Abschnitt über verkettete Listen (Abschnitt 4.8.9, »Fallbeispiel: Verkettete Listen«) wurde bereits erwähnt, dass die Implementierung nur auf Datenobjekte angewandt werden konnte, für die sie programmiert wurden. Im Beispiel waren dies Objekte der Klasse `Daten`. Würden Sie diese verkettete Liste jetzt mit anderen Datenobjekten verwenden wollen, müssten Sie den Code zu den verketteten Listen anpassen, was in der Praxis meistens mehrere Zeilen Code sind, wie es das Beispiel zu den verketteten Listen demonstriert.

Mit den Klassen-Templates haben Sie nun die Möglichkeit, die verkettete Liste so umzuschreiben, dass Sie diese völlig unabhängig von den Datenobjekten verwenden können.

[»] **Hinweis**

In der Praxis werden Sie wohl kaum verkettete Listen in C++ selbst schreiben. Solche immer wiederkehrenden und häufig benötigten Aufgaben bietet Ihnen die STL-Bibliothek. Auf der anderen Seite sind verkettete Listen auch immer eine Art Entwicklungsstufe vom Anfänger zum Profi. In Schulen und Universitäten werden verkettete Listen zudem immer wieder gerne für Prüfungsaufgaben verwendet.

5.2.1 Definition

Wie die Funktions-Templates beginnt auch ein Klassen-Template mit dem Präfix `template <class T>` und der anschließenden Klassendefinition:

```
template <class T>
class KlassenName {
   // ...
};
```

Hier definieren Sie eine Klasse mit dem Namen `KlassenName<T>`. Der Parameter `T` steht für einen beliebigen Typ. Die Angaben `T` und `KlassenName<T>` werden in der Klassendefinition wie normale Typen verwendet.

Die erste Verwirrung dürfte entstehen, weil hier von `T` und `KlassenName<T>` die Rede ist. Die Unterscheidung ist wichtig im richtigen Geltungsbereich. So kann innerhalb des Geltungsbereichs einer Klasse `KlassenName` statt `KlassenName<T>` angegeben werden. Die Angabe `KlassenName<T>` ist der Datentyp, und die Angabe ohne `T` ist der Template-Name.

Bezogen auf die verkettete Liste, sieht das Template der Basisklasse `Knoten` folgendermaßen aus:

```
template <class T>
class Knoten {
public:
   Knoten() {}
   virtual ~Knoten() {}
   virtual Knoten* einfuegen( T* d ) = 0;
   virtual void anzeigen() = 0;
};
```

Mit diesem Klassen-Template wird der Datentyp `Knoten<T>` definiert. Der Parameter `T` ist hierbei der Typ des Objekts, der in der Liste eingefügt werden kann – was im vorherigen Beispiel (siehe Abschnitt 4.8.9, »Fallbeispiel: Verkettete Listen«) zu den verketteten Listen ein Objekt der Klasse `Daten` war (und theoretisch auch hier wieder sein kann).

Es soll auch gleich erwähnt werden, dass es ebenfalls möglich ist, wie bei den Funktions-Templates mehrere Typparameter zu definieren. Hier eine solche Definition mehrerer Typparameter:

```
template <class T1, class T2>
class KlassenName {
   // ...
};
```

Hiermit haben Sie eine Klasse `KlassenName<T1, T2>` definiert. Die Parameter `T1` und `T2` stehen für einen bestimmten Typ.

5.2.2 Methoden von Klassen-Templates definieren

Die Methoden eines Klassen-Templates mit dem Parameter `T` werden, wie die Klasse selbst, über den Typ `T` parametrisiert. Die Definition einer solchen Methode selbst stellt somit wiederum ein Funktions-Template dar. Die Definition außerhalb der Klassen-Templates hat gewöhnlich folgende Syntax:

```
template <class T>
void KlassenName<T>::methodenName( Parameter ) {
   // ...
}
```

Hiermit wurde die Methode `methodenName` der Klasse `KlassenName<T>` definiert.

Bezogen auf die abgeleitete Klasse `AllgemeinerKnoten`, die zuvor noch »Klassen-Template-gerecht« zubereitet werden muss:

```
// ------ Abgeleitete Klasse AllgemeinerKnoten ------------
// Diese Klasse ist für die eigentliche Verwaltung der Daten
// verantwortlich
```

```
template <class T>
class AllgemeinerKnoten : public Knoten<T> {
private:
   T* daten;
   Knoten<T>* next;
public:
   // Konstruktor
   AllgemeinerKnoten(T* d, Knoten<T>* n):daten(d), next(n) {
      // Zu Debug- bzw. Verständniszwecken ggf. entfernen
      cout << "Objekt [AllgemeinerKnoten] erzeugt\n";
   }
   // Destruktor
   ~AllgemeinerKnoten() {
      delete next;
      delete daten;
   }
   // Implizit virtual
   Knoten<T>* einfuegen( T* d );
   // Implizit virtual
   void anzeigen();
};
```

Dadurch sehen die Methoden `einfuegen()` und `anzeigen()` des Typs `AllgemeinerKnoten<T>` folgendermaßen aus:

```
template <class T>
Knoten<T>* AllgemeinerKnoten<T>::einfuegen( T* d ) {
   // Wir sortieren aufwärts - kleiner Wert vor großem Wert
   int ret = daten->vergleichen( *d );
   switch( ret ) {
      case 0:
      case 1: {
         // Neue Daten vor dem aktuellen einordnen
         AllgemeinerKnoten<T>* dKnoten =
            new AllgemeinerKnoten<T>( d, this );
         return dKnoten;
      }
      case -1:
         // größer als das aktuelle Element -
         // weiter zum nächsten Knoten
         next = next->einfuegen( d );
         return this;
   }
   return this;
}
```

```
template <class T>
void AllgemeinerKnoten<T>::anzeigen() {
   daten->anzeigen();
   next->anzeigen();
}
```

Die Methode `einfuegen()` kann nun Objekte vom Typ `T` in der verketteten Liste einfügen.

Hierzu gleich noch der komplette Code der Header-Datei *lliste.h*, der jetzt komplett auf Klassen-Template »getrimmt« wurde und nicht mehr von einem Datenobjekt abhängig ist:

```
// lliste.h
#include <iostream>
#ifndef _LLISTE_H_
#define _LLISTE_H_

// abstrakte Basisklassen-Template Knoten<T>
// Alle abgeleiteten Klassen müssen "einfuegen"
// und "anzeigen" redefinieren
template <class T>
class Knoten {
public:
   Knoten() {}
   virtual ~Knoten() {}
   virtual Knoten* einfuegen( T* d ) = 0;
   virtual void anzeigen() = 0;
};

// ------ Abgeleitete Klasse AllgemeinerKnoten ------------
// Diese Klasse ist für die eigentliche Verwaltung der Daten
// verantwortlich
template <class T>
class AllgemeinerKnoten : public Knoten<T> {
private:
   T* daten;
   Knoten<T>* next;
public:
   // Konstruktor
   AllgemeinerKnoten(T* d, Knoten<T>* n):daten(d), next(n) {
      // Zu Debug- bzw. Verständniszwecken ggf. entfernen
      cout << "Objekt [AllgemeinerKnoten] erzeugt\n";
   }
   // Destruktor
   ~AllgemeinerKnoten() {
```

```
      delete next;
      delete daten;
   }
   // Implizit virtual
   Knoten<T>* einfuegen( T* d );
   // Implizit virtual
   void anzeigen();
};

// Die wichtigsten Methoden in diesem Programm
template <class T>
Knoten<T>* AllgemeinerKnoten<T>::einfuegen( T* d ) {
   // Wir sortieren aufwärts - kleiner Wert vor großem Wert
   int ret = daten->vergleichen( *d );
   switch( ret ) {
      case 0:
      case 1: {
         // Neue Daten vor dem aktuellen einordnen
         AllgemeinerKnoten<T>* dKnoten =
            new AllgemeinerKnoten<T>( d, this );
         return dKnoten;
      }
      case -1:
         // größer als das aktuelle Element -
         // weiter zum nächsten Knoten
         next = next->einfuegen( d );
         return this;
   }
   return this;
}

template <class T>
void AllgemeinerKnoten<T>::anzeigen() {
   daten->anzeigen();
   next->anzeigen();
}

// ------ Abgeleitete Klasse EndKnoten -------------------
// Der EndKnoten dient im Grunde nur als Endpunkt der Liste
template <class T>
class EndKnoten : public Knoten<T> {
public:
   EndKnoten() {
     // Zu Debug- bzw. Verständniszwecken ggf. entfernen
     cout << "Objekt [EndKnoten] erzeugt\n";
   }
```

```
   ~EndKnoten() {}
   // Implizit virtual
   Knoten<T>* einfuegen( T* d );
   // Implizit virtual
   void anzeigen() { };
};

// Daten werden immer vor dem Ende eingefügt
template <class T>
Knoten<T>* EndKnoten<T>::einfuegen( T* d ) {
   AllgemeinerKnoten<T>* daten =
      new AllgemeinerKnoten<T>(d, this);
   return daten;
}

// ------ Abgeleitete Klasse AnfangsKnoten ---------------
// Knoten, der "nur" immer auf das erste Element der
// Liste zeigt
template <class T>
class AnfangsKnoten : public Knoten<T> {
private:
   // ... zeigt immer auf das erste Element
   Knoten<T>* next;
public:
   // Konstruktor
   AnfangsKnoten<T>() {
      // ... gleich auch einen Endknoten erzeugen
      next = new EndKnoten<T>;
      // Zu Debug- bzw. Verständniszwecken ggf. entfernen
      cout << "Objekt [AnfangsKnoten] erzeugt\n";
   }
   ~AnfangsKnoten() {}
   // Implizit virtual
   Knoten<T>* einfuegen( T* d );
   // Implizit virtual
   void anzeigen();
};

template <class T>
Knoten<T>* AnfangsKnoten<T>::einfuegen( T* d ) {
   // Am Anfang kommen keine Daten rein, daher an den
   // nächsten Knoten weiterreichen
   next = next->einfuegen(d);
   return this;
}
```

```
template <class T>
void AnfangsKnoten<T>::anzeigen() {
   next->anzeigen();
}

// ------ Unabhängige Klasse Liste ------------------
template <class T>
class Liste {
private:
   AnfangsKnoten<T> *anfang;
public:
   // Bei Anlegen gleich ein Objekt "Anfangsknoten" erzeugen
   // Der Konstruktor von "AnfangsKnoten" erzeugt wiederum
   // ein Objekt "EndKnoten", auf das dieser gleich zeigt.
   Liste() { anfang = new AnfangsKnoten<T>; }
   ~Liste() { delete anfang; }
   void einfuegen( T* d ) {
      anfang->einfuegen(d);
   }
   void alles_anzeigen() {
      anfang->anzeigen();
   }
};
#endif
```

[»] **Hinweis**

Auch wenn der Code hier nicht mehr abhängig vom Typ ist, so müssen Sie dennoch Objekte verwenden, die die Methoden `vergleichen()` und `anzeigen()` enthalten.

5.2.3 Klassen-Template generieren (Instantiierung)

Mit dem Klassen-Template zu den verketteten Listen haben Sie jetzt eine Schablone erstellt, aus der Klassen für einen noch festzulegenden Typ generiert werden können. Diesen Typ müssen Sie dem Template als Argument übergeben. Ist dies geschehen, ist das Klassen-Template instantiiert.

Wie schon bei den Funktions-Templates erfolgt die Instantiierung implizit durch den Compiler, wenn das Klassen-Template zum ersten Mal verwendet wird. Im Allgemeinen ist dies bei der ersten Deklaration eines Objekts vom Typ eines Klassen-Templates der Fall:

```
Liste<T> testListe;    // Implizite Instantiierung
```

Hiermit wird zuerst das Klassen-Template `Liste<T>` generiert und anschließend ein Objekt des Typs erzeugt. Bei der Generierung eines Klassen-Templates wird der benötigte Maschinencode für die Methoden mit den Funktions-Templates

erzeugt. Jedes Auftreten des Parameters `T` wird durch den entsprechenden Typ des Template-Arguments ersetzt. Natürlich bedeutet dies auch, dass jedes andere Template-Argument `T` auch einen anderen Maschinencode erhält.

Hinweis [«]

Hier soll nicht der Eindruck entstehen, dass sich das Generieren von Klassen-Templates auf Klassen beschränkt. Es ist selbstverständlich auch möglich, für `T` einfache Basisdatentypen zu verwenden.

Um wieder auf das Beispiel der verketteten Listen zurückzukommen, werden zur Demonstration zwei Klassen verwendet – zum einen die bereits bekannte Klasse `Daten`, und zum anderen die Klasse `Temperatur`. Zuerst nochmals der Code der Klasse `Daten`, in der Header-Datei *daten.h* verpackt:

```
// daten.h
#include <iostream>
using namespace std;
#ifndef _DATEN_H_
#define _DATEN_H_

class Daten {
private:
   int iwert;
public:
   // Konstruktor
   Daten( int iVal ): iwert(iVal) {
      // Zu Debug- bzw. Verständniszwecken ggf. entfernen
      cout << "Objekt [Daten] erzeugt\n";
   }
   // Destruktor
   ~Daten( ) {}
   int vergleichen( const Daten& );
   void anzeigen() const;
};

// Zum Vergleichen der Eigenschaften zweier Objekte
int Daten::vergleichen( const Daten& d ) {
   if( iwert > d.iwert ) { return 1; };
   if( iwert < d.iwert ) { return -1;  };
   return 0;
}

void Daten::anzeigen() const {
   cout << iwert << "\n";
```

```
}
#endif
```

Jetzt noch die Klasse `Temperatur`, zusammengefasst in der Header-Datei *temperatur.h*:

```
// temperatur.h
#include <iostream>
#include <cstring>
using namespace std;
#ifndef _TEMPERATUR_H_
#define _TEMPERATUR_H_

class Temperatur {
private:
   int grad;
   char monat[20];
public:
   // Konstruktor
   Temperatur( int iVal, char* m="" ): grad(iVal){
      // Zu Debug- bzw. Verständniszwecken ggf. entfernen
      strncpy( monat, m, 20 );
      cout << "Objekt [Temperatur] erzeugt\n";
   }
   // Destruktor
   ~Temperatur( ) {}
   int vergleichen( const Temperatur& );
   void anzeigen() const;
};

// Zum Vergleichen der Eigenschaften zweier Objekte
int Temperatur::vergleichen( const Temperatur& d ) {
   if( grad > d.grad ) { return 1; };
   if( grad < d.grad ) { return -1;  };
   return 0;
}

void Temperatur::anzeigen() const {
   cout << monat << " : " << grad << "\n";
}
#endif
```

Die Instantiierung (der beiden Klassen) über das Klassen-Template sieht folgendermaßen aus:

```
Liste<Temperatur> elemente1;
Liste<Daten> elemente2;
```

Hiermit wird jeweils ein Klassen-Template `Liste<Temperatur>` und `Liste <Daten>` generiert. Anschließend wird jeweils ein Objekt des Typs erzeugt. Die Eingabe soll über eine eigene Funktion realisiert werden. Zum Auffrischen wollen wir auch hierzu ein Funktions-Template verwenden:

```
template <class T>
void input( Liste<T>& data ) {
     input_func( data );
}
```

Dieses Funktions-Template können Sie nun aus dem Hauptprogramm mit

```
input(elemente1);
input(elemente2);
```

aufrufen; Sie müssen sich auf diese Weise nicht mehr um den Typ der Eingabe kümmern. In der Funktion `input()` wird dann eine dem Typ entsprechende Funktion `input_func()` generiert. Hierzu das komplette Hauptprogramm:

```
// main.cpp
#include "daten.h"
#include "temperatur.h"
#include "lliste.h"

void input_func( Liste<Temperatur>& tempList );
void input_func( Liste<Daten>& datenList );
template <class T> void input( Liste<T>& data );

int main( void ) {
   // Instantiierung
   Liste<Temperatur> elemente1;
   Liste<Daten> elemente2;
   input(elemente1);   // Liste<Temperatur>
   input(elemente2);   // Liste<Daten>
   elemente1.alles_anzeigen();
   elemente2.alles_anzeigen();
   return 0;
}

// Funktion zum Einlesen der Eigenschaften für Temperatur
void input_func( Liste<Temperatur>& tempList ) {
   Temperatur* daten;
   int grad;
   char monat[20];
   for(;;) {
      cout << "Monat eingeben : ";
```

```
      if( (!(cin >> monat)) )
         strcpy( monat, "keine Angabe");
      cout << "Temperatur eingeben (99=Ende) : ";
      if( (!(cin >> grad)) || grad == 99 )
         break; // Ende
      daten = new Temperatur(grad, monat);
      tempList.einfuegen(daten);
   }
}

// Funktion zum Einlesen der Eigenschaften für Daten
void input_func( Liste<Daten>& datenList ) {
   Daten* daten;
   int iwert;
   for(;;) {
      cout << "Wert eingeben (0=Ende) : ";
      if( (!(cin >> iwert)) || iwert == 0 )
         break; // Ende
      daten = new Daten(iwert);
      datenList.einfuegen(daten);
   }
}

// Funktions-Template
template <class T>
void input( Liste<T>& data ) {
     input_func( data );
}
```

Das Programm bei der Ausführung:

```
Objekt [EndKnoten] erzeugt
Objekt [AnfangsKnoten] erzeugt
Objekt [EndKnoten] erzeugt
Objekt [AnfangsKnoten] erzeugt
Monat eingeben : Januar
Temperatur eingeben (99=Ende) : 10
Objekt [Temperatur] erzeugt
Objekt [AllgemeinerKnoten] erzeugt
Monat eingeben : Februar
Temperatur eingeben (99=Ende) : 8
Objekt [Temperatur] erzeugt
Objekt [AllgemeinerKnoten] erzeugt
Monat eingeben : März
Temperatur eingeben (99=Ende) : 15
```

```
Objekt [Temperatur] erzeugt
Objekt [AllgemeinerKnoten] erzeugt
Monat eingeben : 0
Temperatur eingeben (99=Ende) : 99

Wert eingeben (0=Ende) : 6
Objekt [Daten] erzeugt
Objekt [AllgemeinerKnoten] erzeugt
Wert eingeben (0=Ende) : 4
Objekt [Daten] erzeugt
Objekt [AllgemeinerKnoten] erzeugt
Wert eingeben (0=Ende) : 8
Objekt [Daten] erzeugt
Objekt [AllgemeinerKnoten] erzeugt
Wert eingeben (0=Ende) : 0

Februar : 8
Januar : 10
März : 15

4
6
8
```

5.2.4 Weitere Template-Parameter

Es wurde bereits erwähnt, dass ein Klassen-Template weitere Parameter enthalten kann. Ein solcher Parameter muss aber nicht zwangsläufig ein Typname sein. Sie können hierzu (was in der Praxis auch häufig der Fall ist) »gewöhnliche« Datentypen verwenden, auch Zeiger und Referenzen:

```
template <class T, int n>
class Array {
   // ...
};
```

Hiermit haben Sie eine Klasse vom Typ `Array<T, n>` definiert. Diese Klasse ist mit dem Typ `T` und einer Ganzzahl `n` parametrisiert. In diesem Beispiel wird der Parameter `n` dazu verwendet, die Größe des Arrays vom Typ `T` festzulegen. Damit ist zum Zeitpunkt der Generierung des Klassen-Templates die Anzahl der Elemente bekannt, die das Array haben wird.

Dennoch gibt es zwei Regeln zum Einsatz von Template-Parametern, die keine Typparameter sind:

1. Der Wert eines Template-Parameters darf niemals verändert werden.
2. Der Template-Parameter darf niemals ein Gleitpunkttyp sein. Allerdings kann man hierbei die Hintertür verwenden und stattdessen Zeiger oder Referenzen auf Gleitpunkttypen definieren.

Bei der Verwendung von Methoden solcher Klassen-Templates mit mehreren Parametern müssen Sie dies auch bei der Definition berücksichtigen:

```
template <class T, int n>
T &Array<T, n>::methode (parameter) {
   // ...
}
```

Wenn Sie jetzt ein Klassen-Template instantiieren wollen, müssen Sie für jeden entsprechenden Template-Parameter ein Argument angeben:

```
Array<char, 20> str1;
Array<int, 5> integers;
```

Hier werden eine Klasse `Array<char, 20>` und eine Klasse `Array<int, 5>` generiert. Danach wird ein entsprechendes Objekt dieser Klassen erzeugt. Dies wären ein Array mit 20 `char`-Werten und ein zweites mit 5 `int`-Werten.

Auch bei der Instantiierung gibt es für die Argumente von Klassen-Templates, die keine Typnamen sind, einige Regeln zu beachten:

- Der Parameter muss ein konstanter Ausdruck sein, außer es handelt sich um eine Referenz.
- Als Zeiger kommen nur Adressen eines Objekts oder einer Funktion mit globalem Geltungsbereich in Frage.
- Verwenden Sie eine Referenz als Parameter, so muss das Argument ein Objekt sein, das `global` oder `static` definiert wurde.

Bezogen auf den hier erwähnten zweiten Punkt, bedeutet dies, dass eine Instantiierung einer String-Konstante fehlschlägt, weil ihr Geltungsbereich nicht global, sondern statisch ist. Zum Beispiel führt folgende Instantiierung zu einer Fehlermeldung des Compilers:

```
template <class T, char* str>
class KlassenName {
   // ...
};
...
int main() {
   KlassenName<long, "long"> Clong;  // Fehler!!!
   ...
}
```

Erst mit einer globalen Definition des Strings können Sie dieses Template-Argument einsetzen:

```
template <class T, char* str>
class KlassenName {
   // ...
};
...
char str[] = "long";
...
int main() {
   KlassenName<long, str> Clong;
   ...
}
```

Hierzu ein komplettes Beispiel des Klassen-Templates `Array`, das außerdem auch sehr schön die Überladung des []-Operators – siehe Abschnitt 4.5.6, »Überladen des Indexoperators '[]' (Arrays überladen)« – demonstriert:

```
// Array1.cpp
#include <iostream>
#include <cstdlib>
using namespace std;

template <class T, int n>
class Array {
private:
   T array[n];
public:
   int get_size () const {
      return n;
   }
   T& operator[] (int i);
};

template <class T, int n>
T &Array<T, n>::operator[] (int i) {
   if(i < 0 || i >= n ) {
      cout << "Array-Überlauf!!!\n";
      exit(1);
   }
   return array[i];
}
```

```
int main( void ) {
   Array<char, 20> str1;
   Array<int, 5> integers;
   cout << "Wie heißen Sie: ";
   cin.getline( &str1[0], str1.get_size() );
   cout << "Hallo ";
   for(int i=0; str1[i] != '\0'; i++) {
      cout << str1[i];
   }
   cout << "\n";
   for( int i=0; i < integers.get_size(); i++ ) {
      cout << i+1 << ". Wert: ";
      cin >> integers[i];
   }
   cout << "Die Zahlen, die eingegeben wurden ...\n";
   for( int i=0; i < integers.get_size(); i++ ) {
      cout << i+1 << ". Wert: " << integers[i] << "\n";
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Wie heißen Sie: Jürgen Wolf
Hallo Jürgen Wolf
1. Wert: 12
2. Wert: 34
3. Wert: 56
4. Wert: 78
5. Wert: 90
Die Zahlen, die eingegeben wurden ...
1. Wert: 12
2. Wert: 34
3. Wert: 56
4. Wert: 78
5. Wert: 90
```

5.2.5 Standardargumente von Templates

Wie schon bei herkömmlichen Funktionen können Sie auch bei den Template-Parametern Standardargumente verwenden. Diese können Sie entweder bei der Definition eines Templates oder bei der ersten Deklaration eines Templates in einem Modul angeben. Das Array-Template könnten Sie mit den folgenden Standardargumenten definieren:

```
template <class T=char, int n=50>
class Array {
   // ...
};
```

Bei der Instantiierung eines Klassen-Templates oder eines Funktions-Templates können dann die entsprechenden Argumente weggelassen werden. Beispielsweise erreichen Sie mit

```
// Array mit 50 char-Elemente
Array<> str1;
// Fehler !!! - Typ Array gibt es nicht
Array str2;
```

dass nur mit der Angabe der spitzen Klammern ein Array mit 50 `char`-Elementen erzeugt wird. Wie bei den Funktionen können Sie auch bei den Templates andere Werte, als vorgegeben sind, übergeben – aber auch hier gelten wieder dieselben Regeln wie bei den Standardargumenten von Funktionen:

- Wenn Sie einen Template-Parameter mit einem Standardargument versehen, müssen Sie auch alle nochfolgenden Template-Parameter mit einem Standardargument versehen.
- Wenn Sie bei der Instantiierung eines Templates ein Argument weglassen, für das ein Standardargument existiert, dann müssen Sie auch die nachfolgenden Argumente weglassen.

Hierzu einige Möglichkeiten, wie Sie das Array-Template

```
template <class T=char, int n=50>
class Array {
   // ...
};
```

noch instantiieren können:

```
// char-Array mit 50 Elementen
Array<> str1;
// int-Array mit 50 Elementen
Array<int> integers;
// long-Array mit 10 Elementen
Array<long, 10> longs;

// Nicht möglich --- Fehler !!!
Array<10> str2;
// ... wenn, dann so ...
Array<char, 10> str2;  // 10 char-Elemente
```

5.2.6 Explizite Instantiierung

Neben der impliziten Instantiierung eines Templates bei der Definition eines Objekts ist es auch möglich, ein Klassen-Template (wie auch ein Funktions-Template) explizit zu instantiieren. So kann das Funktions-Template

```
template <class T>
void func( T val[], int len ) {
   // ...
}
```

folgendermaßen explizit instantiiert werden:

```
template void func( int val[], int len );
```

Hier wurde das Funktions-Template `func()` explizit für den Datentyp `int` instantiiert. Dies bedeutet allerdings in der Praxis, dass diese Funktion auch wirklich generiert wird, egal, ob sie nun aufgerufen wird oder nicht. Das ist auch der Unterschied zwischen der impliziten und der expliziten Instantiierung. Bei einer expliziten Instantiierung wird die Funktion immer generiert – bei einer impliziten hingegen nur, wenn die Funktion in dieser Übersetzungseinheit aufgerufen wird.

Gleiches gilt auch für ein Klassen-Template. Wenn Sie das Klassen-Template

```
template <class T>
class Array {
   // ...
};
```

mit folgender Anweisung explizit instantiieren würden

```
template class Array<char>;
```

wird tatsächlich eine Klasse `Array` mit sämtlichen Methoden für den Typ `char` instantiiert, unabhängig davon, ob diese Klasse anschließend auch verwendet wird.

Die explizite Instantiierung wird gewöhnlich für die Erstellung von Bibliotheken verwendet, wenn Klassen- bzw. Funktions-Templates nur für andere Übersetzungseinheiten zur Verfügung gestellt werden.

Wurde also ein Klassen- oder Funktions-Template explizit instantiiert, so müssen Sie in einer anderen Übersetzungseinheit (Modul) nur eine einfache Deklaration verwenden, um die Funktion bzw. Klasse zu verwenden. Eine solche Deklaration stellt man gewöhnlich in eine Header-Datei wie zum Beispiel:

```
// Enthält die Definition des Klassen-Templates Array
#include "array.h"

// Explizite Instantiierungen für int und char
template class Array<int>;
template class Array<char>;
```

Durch diese Vorgehensweise können Sie verhindern, dass einzelne Klassen- bzw. Funktions-Templates in verschiedenen Modulen mehrmals generiert werden (wobei der Linker diese »Verdoppelung« später wieder entfernt). Und der Teil eines Klassen-Templates, in dem sich die Definition von Methoden und statischen Elementen befindet, muss nur in dem Modul vorhanden sein, in dem die Instanzen der Klassen generiert werden.

5.3 STL (Standard Template Library)

Basierend auf diesem mächtigen Feature der Templates, baut auch STL (*Standard Template Library*) auf, die ursprünglich als eine Extrabibliothek bei Hewlett-Packard entwickelt wurde und mittlerweile ein Teil des C++-Standards geworden ist. Der Schwerpunkt dieser Bibliothek liegt auf häufig verwendeten und benötigten Datenstrukturen (für Container – Behälter) und Algorithmen (um das Rad nicht immer wieder neu zu erfinden).

Neben dem Vorteil, dass STL zum C++-Standard gehört und somit einfacher portierbar ist, haben Templates im Allgemeinen den Vorteil, dass die Auswertung der Templates zur Compiler-Zeit stattfindet und somit die Laufzeit des Programms nicht beeinflusst, wie dies etwa bei polymorphen Funktionen der Fall ist, wenn die Generizität mit Vererbung implementiert wurde. Außerdem bringt dies auch eine Typsicherheit mit sich, weil das Template zur Compiler-Zeit aufgelöst wird.

Zu den zentralen Bestandteilen gehören Container, Algorithmen und Iteratoren. Die Container sind »Behälter«, die für Objekte verschiedenster Klassen verwendet werden können (dank Templates). Der Iterator ist ein Objekt, das wie ein Zeiger auf einem Container durchlaufen wird. Damit ist es möglich, die einzelnen Elemente oder Teilbereiche von Containern anzusprechen. Algorithmen wiederum arbeiten mit Containern, indem diese auf den dazugehörenden Iterator zugreifen. Auf das Konzept von STL wird im nächsten Abschnitt eingegangen.

Hinweis **[«]**

STL ist unglaublich mächtig und umfangreich. Daher sei darauf hingewiesen, dass hier lediglich eine umfassende Einführung in STL gegeben werden kann.

[»] **Hinweis**

Sämtliche Bezeichner der Container-Bibliothek sind im Standard-Namensbereich `std` enthalten – ebenso wie alle Algorithmen und Klassen von STL.

5.3.1 Konzept von STL

Wie bereits erwähnt, basiert das Konzept von STL auf den Komponenten Container, Iterator und Algorithmen, deren Bedeutung und Zusammenwirken jetzt näher erläutert werden soll.

Container

In STL gibt es verschiedene Arten von Containern, die ebenfalls als Klassen-Template implementiert sind. Container speichern Objekte eines bestimmten Typs und verwalten diese zur Laufzeit dynamisch. Dem Programmierer bleibt es dann überlassen, ob er das Objekt als Wert oder als Referenz ablegt. STL unterscheidet hierbei zwischen sequentiellen und assoziativen Containern. Bei sequentiellen Containern sind die Objekte linear angeordnet und können über ihre Position im Container angesprochen werden. Assoziative Container sind Objekte, die in einer Baumstruktur vorliegen – hierbei erfolgt der Zugriff über Schlüssel.

Zu den sequentiellen Containern, die STL anbietet, gehören:

- Listen
- Vektoren (Arrays)
- Deques (*double-ended queues*)
- Stacks (nach dem LIFO-Prinzip – *Last In First Out*)
- Queues (Warteschlangen nach dem FIFO-Prinzip – *First In First Out*)
- Priority-Queues (Warteschlangen mit Prioritäten)

Zu den assoziativen Containern zählen:

- Sets (Suchen und Sortieren von Objekten)
- Maps (Paare aus Objekt und Schlüssel)
- Bitsets (wird zur Manipulation von Bits verwendet)

Auf die einzelnen Funktionen und Verwendungsweisen dieser Container wird auf den kommenden Seiten noch eingegangen.

Iteratoren

Iteratoren sind so etwas wie abstrakte Zeiger und die Grundlage zum Navigieren in den Containern. Iteratoren werden also für den Zugriff auf ein Container-Ele-

ment verwendet. Wenn Sie einen Iterator `position` deklariert haben und `++position` verwenden, wird der Iterator auf das nächste Element im Container gesetzt. Mit `--position` setzen Sie den Iterator auf den Vorgänger des aktuellen Elements. Mit dem Ausdruck `*position` erhalten Sie das aktuelle Objekt an der Position `position` zurück.

Algorithmen

Die Template-Algorithmen wiederum arbeiten mit den Iteratoren, die auf Container zugreifen. Somit kann man das ganze Konzept in einem Satz zusammenfassen: Container stellen Iteratoren zur Verfügung, und die Algorithmen benutzen diese.

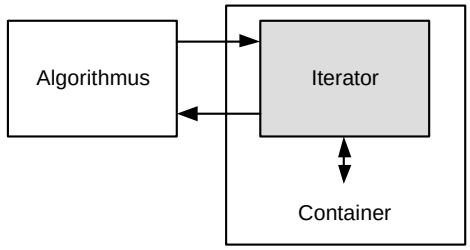

Abbildung 5.1 Algorithmen, Container und Iteratoren im Zusammenspiel

Dieses Konzept möchte ich Ihnen zunächst einmal ohne STL demonstrieren. Hierzu soll ein einfaches `int`-Array mit beliebigen Werten versehen werden. In diesem Array soll mit einer Funktion `find()` ein gewünschter Wert gefunden werden. Wo wird hier das Konzept von STL implementiert, werden Sie sich fragen. Als Container dient das `int`-Array, und der Iterator ist ein Zeiger auf das Array. Der Algorithmus `find()` benutzt diesen Iterator.

```
// stl1.cpp
#include<iostream>
using namespace std;

// Der Iterator (Zeiger auf int)
typedef int* myIterator;

// Der Algorithmus (Prototyp des Algorithmus)
myIterator find( myIterator begin, myIterator end,
                 const int& ival );
// Container und Iterator initialisieren
void init( int* Container, int count );

int main() {
```

```
   const int cnt = 50;
   // Der Container (einfaches int-Array)
   int Container[cnt];
   // Die Iteratoren
   // Zeiger auf den Anfang des Containers
   myIterator begin = Container;
   // Zeiger hinter das letzte Element des Containers
   myIterator end = Container + cnt;
   init( &Container[0],  cnt );
   int num = -1;
   while( num != 0 ) {
      cout << "Nummer suchen (0=Ende) : ";
      if( !(cin >> num) ) {
         break; // Fehlerhafte Eingabe
      }
      if( num != 0  ) {
         // Algorithmus verwendet Iterator
         myIterator pos = find( begin, end, num );
         if( pos != end ) {
            cout << num << " an Pos. "
                 << (pos-begin) << "\n";
         }
         else {
            cout << num << " ist nicht vorhanden!!\n";
         }
      }
   }
   return 0;
}

// Implementierung des Algorithmus
myIterator find( myIterator begin, myIterator end,
                                   const int& ival) {
   while(begin != end && *begin != ival) {
      ++begin; // nächste Position
   }
   return begin;
}

void init( int* Container, int count ) {
   // Container mit Werten initialisieren
   for(int i = 0; i < count; i++) {
      Container[i] = i*i;
   }
}
```

Das Programm bei der Ausführung:

```
Nummer suchen (0=Ende) : 36
36 an Pos. 6
Nummer suchen (0=Ende) : 100
100 an Pos. 10
Nummer suchen (0=Ende) : 2000
2000 ist nicht vorhanden!!
Nummer suchen (0=Ende) : 0
```

Dieses Beispiel ist noch immer abhängig vom Typ programmiert. Hierbei könnten Sie jetzt Templates verwenden, aber genau bei solchen immer wiederkehrenden Aufgaben greift STL ein. Ohne jetzt bereits genauer darauf einzugehen, soll für dasselbe Beispiel die STL-Bibliothek verwendet werden. Bei genauerem Hinsehen werden Sie die drei Konzepte Container, Iterator und Algorithmus wiedererkennen.

```
// stl2.cpp
#include <iostream>
#include <vector>     // vector
#include <algorithm>  // für find()
using namespace std;

int main() {
   const int cnt = 50;
   // Der Container
   vector<int> Container(cnt);
   for(int i = 0; i < cnt; i++) {
      Container[i] = i*i;
   }
   int num = -1;
   while( num != 0 ) {
      cout << "Nummer suchen (0=Ende) : ";
      if( !(cin >> num) ) {
         break; // Fehlerhafte Eingabe
      }
      if( num != 0  ) {
         // Der Iterator verwendet den Algorithmus
         vector<int>::iterator pos = find (
            Container.begin(), Container.end(), num );
         if( pos != Container.end() ) {
            cout << num << " an Pos. "
                 << (pos-Container.begin()) << "\n";
         }
         else {
```

```
            cout << num << " ist nicht vorhanden!!\n";
         }
      }
   }
   return 0;
}
```

Auf die einzelnen Container, Algorithmen und deren Methoden wird noch extra eingegangen. Die Vorteile lassen sich wohl schon erahnen. Im Prinzip müssen nur noch einzelne Anweisungen von STL zusammengebaut werden, damit diese miteinander harmonieren. Zum einen werden die Programme dadurch erheblich kürzer, und zum anderen (der Hauptvorteil) werden viele Programmierfehler vermieden, da es sich bei STL um eine sehr gut getestete und ausgereifte Bibliothek handelt. Das Erstellen von Programmen benötigt daher erheblich weniger Zeit.

5.3.2 Hilfsmittel (Hilfsstrukturen)

Neben den Containern, Iteratoren und Algorithmen stellt STL auch noch verschiedene Hilfsstrukturen zur Verfügung, zum Beispiel Paar-Strukturen (`pair`), die für assoziative Listen von Bedeutung sind, oder Funktionsobjekte. Hier eine kurze Beschreibung zu den Hilfsmitteln von STL.

pair

In der STL-Bibliothek finden Sie einen definierten Datentyp `pair` (für Paar), mit dem Sie einen Typ definieren können, der zwei Daten zusammenfassen kann, wobei die beiden Datentypen beliebig sein können (also auch – und vor allem – Objekte von Klassen). `pair` ist in der Header-Datei <*utility*> definiert und muss daher auch bei den Beispielen, die `pair` verwenden, eingebunden sein.

`pair` ist übrigens auch eine Antwort auf die Frage, wie man in C++ mehrere Werte aus einer Funktion zurückgeben kann.

[»] Hinweis

Beachten Sie bitte, dass `pair` wie alle anderen Standardbibliotheks-Komponenten im Namensraum `std` gültig ist. Sie verwenden also entweder `using namespace std` oder den Scope-Operator `std::pair`.

Die Syntax zur Definition eines `pair`-Objekts ist im Grunde immer gleich aufgebaut. Nach dem Datentyp `pair` geben Sie die beiden Typen in spitzen Klammern an, die im `pair`-Objekt abgelegt werden sollen. Dahinter folgt der Bezeichner. Folgt hinter dem Bezeichner nichts mehr, werden die beiden Typen (sofern es sich um Basisdatentypen handelt) zunächst mit `0` initialisiert.

```
pair<typ1, typ2> bezeichner;
```

Wer sich fragt, wie ein solcher Datentyp implementiert ist, der kann sich die Deklaration in der Header-Datei <*utility*> ansehen:

```
template <class T1, class T2>
struct pair {
   T1 first;
   T2 second;
   // Konstruktor
   pair(){};
   // initialisiere first mit x und second mit y:
   pair(const T1& x, const T2& y);
   // copy constructor:
   template<class U, class V> pair(const pair<U, V> &p);
};
```

pair-Basisdatentypen

Ein Standardkonstruktor bewirkt, dass die Elemente des jeweiligen Typs initialisiert werden. Hier einige Beispiele, wie Sie `pair`-Objekte mit einfachen Basisdatentypen erstellen können:

```
// pair für char* und long
pair<char*, long> clong;
// Werte an clong zuweisen
clong = pair<char*, long>("long", 1234);
// Gleich bei der Definition mit Werten initialisieren
pair<char*, float> cfloat("float", 111.111);
```

pair-Objekte

Natürlich können Sie in einem `pair`-Objekt auch Objekte verschiedener Klassen abspeichern. Allerdings müssen hier folgende Voraussetzungen erfüllt sein:

- Die Objekte müssen einen Kopierkonstruktor besitzen.
- Die Objekte müssen zuweisbar sein – somit muss der Zuweisungsoperator entsprechend überladen sein.
- Die Objekte müssen vergleichbar sein. Dazu müssen mindestens die Operatoren < und == durch `friend`-Funktionen überladen werden. Die restlichen Vergleichsoperatoren werden automatisch aus diesen beiden überladenen Operatoren gebildet. Das ist so, weil STL im Namensraum `std::rel_ops` Vergleichsoperatoren zur Verfügung stellt, die es ermöglichen, dass in einer Klasse nur die Operatoren == und < definiert sein müssen und dennoch der komplette Satz an Vergleichsoperatoren vorliegt.

Hier eine Klasse, die die eben genannten Anforderungen erfüllt:

```
class A {
private:
   int ival;
public:
   A(int val=0):ival(val){}
   ~A(){}
   // Kopierkonstruktor
   A(const A& Aobj) {
      ival=Aobj.ival;
   }
   // Zuweisungsoperator überladen
   A& operator=(const A& Aobj) {
      ival=Aobj.ival;
   }
   // Überladener <-Operator
   friend bool operator < (const A& Aobj1, const A& Aobj2) {
      if(Aobj1.ival < Aobj2.ival)
         return true;
      else
         return false;
   }
   // Überladener ==-Operator
   friend bool operator==(const A& Aobj1, const A& Aobj2) {
      if( Aobj1.ival == Aobj2.ival )
         return true;
      else
         return false;
   }
};
```

Wollen Sie Objekte nun in einem `pair`-Typ ablegen, müssen Sie nur innerhalb der spitzen Klammern die Klasse des Objekts angeben. Hierbei ist es auch gestattet, Klassen und Basisdatentypen zu mischen. Natürlich können auch Zeiger verwendet werden. Einige weitere `pair`-Beispiele:

```
// 1. Beispiel
pair<A, char*> Ac;
Ac = pair<A, char*>(A(1111), "A-Klasse");

// 2. Beispiel
A aObj(100);
pair<char*, A> cA("A-Klasse", aObj);
```

```
// 3. Beispiel
pair<B, char*> Bc(B(200, 111.111), "B-Klasse");

// 4. Beispiel
pair<A*, B*> dynKlasseAB (new A(111), new B(222, 222.222));
```

Betrachten Sie zunächst das erste Beispiel, das am meisten Rechenaufwand betreibt. Zuerst wird ein leeres Paar `Ac` definiert – wobei der Standardkonstruktor der Klasse `A` ausgeführt wird. In der nächsten Zeile erfolgt eine Zuweisung an das Paar `Ac`, wobei zunächst ein temporäres Objekt erzeugt wird. Dieses temporär erzeugte Objekt wird über den Kopierkonstruktor in ein weiteres `pair`-Objekt gespeichert. Erst jetzt wird der Zuweisungsoperator des Objekts ausgeführt. Natürlich muss am Ende noch der Destruktor des temporären Objekts aufgerufen werden.

Im zweiten Beispiel wird zunächst ein neues Objekt der Klasse `A` erzeugt. Diese wird anschließend einem `pair` bei der Definition zugewiesen, wobei hier der Kopierkonstruktor ausgeführt wird.

Im dritten Beispiel wird gleich bei der Definition von `pair` das Objekt der Klasse definiert. Zunächst wird wieder ein temporäres Objekt erzeugt, das per Kopierkonstruktor übernommen wird. Das temporäre Objekt wird wieder vom Destruktor gelöscht.

Natürlich ist es auch möglich, Zeiger statt Daten oder Objekte in einem `pair` abzulegen, wie es im vierten Beispiel gezeigt wird. Allerdings müssen Sie darauf achten, die Objekte explizit wieder zu löschen, und – das altbekannte Problem – wenn Sie ein solches Objekt kopieren oder zuweisen, zeigen beide Zeiger auf die gleiche Adresse im Speicher, in dem Fall auf das gleiche Objekt.

Zugriff auf »pair«-Elemente

Es wurde bereits gezeigt, dass `pair` als eine öffentliche (`public`) Struktur (`struct`) definiert ist. Somit wird auf die einzelnen Werte dieser Struktur über den Bezeichner, gefolgt vom gewünschten Element, zugegriffen. Das erste `pair`-Element ist über den Namen `first` und das zweite Element über `second` erreichbar. Hier nochmals die Struktur `pair`:

```
template <class T1, class T2>
struct pair {
   T1 first;
   T2 second;
   // Konstruktor
   pair(){};
```

```
   // initialisiere first mit x und second mit y:
   pair(const T1& x, const T2& y);
   // copy constructor:
   template<class U, class V> pair(const pair<U, V> &p);
};
```

Somit sieht der Zugriff in der Praxis wie folgt aus:

```
cout << clong.first << ":" << clong.second << "\n";
cout << cfloat.first << ":" << cfloat.second << "\n";
cout << cA.first << ":" << cA.second.get_ival() << "\n";
```

pair als Rückgabewert von Funktionen

Ein interessantes Feature von `pair` ist die Möglichkeit, zwei Werte aus einer Funktion zurückzugeben. Hierzu ein einfaches Beispiel:

```
inline pair<char*, double>
ret2val(int ival) {
   if( ival <=0 )
      return pair<char*, double>("Fehler", 0.0f);
   else
      return pair<char*, double>("OK", 1.0f );
}
```

Als Rückgabewert liefert diese Funktion ein `pair` aus den Werten `char*` und `double`. Aufrufen können Sie eine solche Funktion wie folgt:

```
pair<char*, double> ret;
ret = ret2val(1);
cout << ret.first << " : " << ret.second << "\n";
```

Hierzu ein komplettes Listing, das alles Beschriebene zu `pair` nochmals in der Praxis demonstriert:

```
// stl3.cpp
#include <iostream>
#include <utility>
using namespace std;

class A {
private:
   int ival;
public:
   A(int val=0):ival(val){}
   ~A(){}
   // Kopierkonstruktor
   A(const A& Aobj) {
```

```
      ival=Aobj.ival;
   }
   // Zuweisungsoperator überladen
   A& operator=(const A& Aobj) {
      ival=Aobj.ival;
   }
   // Überladener <-Operator
   friend bool operator < (const A& Aobj1, const A& Aobj2) {
      if(Aobj1.ival < Aobj2.ival)
         return true;
      else
         return false;
   }
   // Überladener ==-Operator
   friend bool operator==(const A& Aobj1, const A& Aobj2) {
      if( Aobj1.ival == Aobj2.ival )
         return true;
      else
         return false;
   }
   int get_ival() const {
      return ival;
   }
   void set_ival(int val) {
      ival=val;
   }
};

class B {
private:
   long lval;
   float fval;
public:
   B(long val1=0, float val2=0.0f):lval(val1), fval(val2){}
   ~B(){}
   // Kopierkonstruktor
   B(const B& Bobj) {
      lval=Bobj.lval;
      fval=Bobj.fval;
   }
   // Zuweisungsoperator überladen
   B& operator=(const B& Bobj) {
      lval=Bobj.lval;
      fval=Bobj.fval;
   }
```

```
   // Überladener <-Operator
   friend bool operator < (const B& Bobj1, const B& Bobj2) {
      if( (Bobj1.lval < Bobj2.lval) &&
          (Bobj1.fval < Bobj2.fval) )
         return true;
      else
         return false;
   }
   // Überladener ==-Operator
   friend bool operator==(const B& Bobj1, const B& Bobj2) {
      if( (Bobj1.lval == Bobj2.lval) &&
          (Bobj1.fval == Bobj2.fval) )
         return true;
      else
         return false;
   }
   // ... etc. fehlen noch weitere Operatoren
   long get_lval() const {
      return lval;
   }
   float get_fval() const {
      return fval;
   }
};

inline pair<char*, double>
ret2val(int ival) {
   if( ival <=0 )
      return pair<char*, double>("Fehler", 0.0f);
   else
      return pair<char*, double>("OK", 1.0f );
}

int main() {
   pair<char*, long> clong;
   clong = pair<char*, long>("long", 1234);
   pair<char*, float> cfloat("float", 111.111);
   A aObj(100);
   pair<char*, A> cA("A-Klasse", aObj);
   pair<B, char*> Bc(B(200, 111.111), "B-Klasse");

   pair<A, B> KlasseAB(A(123), B(345, 123.456));
   pair<A, B> CopyKlasseAB = KlasseAB;

   pair<A*, B*> dynKlasseAB(new A(111),new B(222, 222.222));
```

```
   pair<char*, double> ret;
   ret = ret2val(1);
   cout << ret.first << " : " << ret.second << "\n";

   if( CopyKlasseAB == KlasseAB ) {
      cout << "Beide Paare sind gleich\n";
   }
   else {
      cout << "Die beiden Paare sind verschieden\n";
   }
   KlasseAB.first.set_ival(11);
   if( CopyKlasseAB == KlasseAB ) {
      cout << "Beide Paare sind gleich\n";
   }
   else {
      cout << "Die beiden Paare sind verschieden\n";
   }
   cout << clong.first << ":" << clong.second << "\n";
   cout << cfloat.first << ":" << cfloat.second << "\n";
   cout << cA.first << ":" << cA.second.get_ival() << "\n";
   cout << Bc.first.get_lval() << ":" << Bc.first.get_fval()
        << ":" << Bc.second << "\n";

   cout << dynKlasseAB.first->get_ival() << " : "
        << dynKlasseAB.second->get_lval() << " : "
        << dynKlasseAB.second->get_fval() << "\n";
   return 0;
}
```

Das Programm bei der Ausführung:

```
OK : 1
Beide Paare sind gleich
Die beiden Paare sind verschieden
long:1234
float:111.111
A-Klasse:100
200:111.111:B-Klasse
111 : 222 : 222.222
```

Vergleichsoperatoren

Es wurde zwar schon bei `pair` erwähnt, aber es sollte hier nochmal extra aufgeführt werden: Für Objekte von selbstdefinierten Typen müssen immer nur die Vergleichsoperatoren == und < selbst definiert werden. Die restlichen Vergleichsoperatoren sind daraus ableitbar:

Operation	Abgeleitet von ...
X!=Y	!(Y==X)
X<=Y	!(Y<X)
X>=Y	!(X<Y)
X>Y	Y<X

Tabelle 5.1 Vergleiche gebildet aus == und <

Damit Sie dieses Feature verwenden können, müssen Sie lediglich die Templates des Namensbereichs `std::rel_ops` einbinden (und natürlich die Header-Datei <*utility*>):

```
#include <utility>
using namespace std::rel_ops;
```

Funktionsobjekte

Viele Funktionen der Algorithmen-Bibliothek und einige Methoden der Container-Klassen erwarten als Argument ein Funktionsobjekt (beispielsweise das zu verwendende Vergleichskriterium). Ein solches Funktionsobjekt kann entweder eine gewöhnliche Funktion sein oder aber das Objekt einer Klasse, für die der Operator `operator()` überladen wurde. Ein solches Objekt kann wie eine Funktion aufgerufen werden. Solche algorithmischen Objekte werden auch als *Funktionsobjekt* oder *Funktor* bezeichnet. Funktoren sind Objekte, die sich zwar wie gewöhnliche Funktionen verhalten, aber alle Eigenschaften von Objekten haben können.

In STL spielt dies besonders bei unären und binären Funktionsobjekten eine wichtige Rolle, weil viele Algorithmen auf diesen aufbauen. Dazu stellt STL in der Header-Datei <*funktional*> zwei Basisklassen, `binary_function` und `unary_function`, für Funktionsobjekte zur Verfügung – womit unäre und binäre Funktionsobjekte vererbt werden können, so dass diese dann von den Algorithmen richtig erkannt werden.

Hierzu ein einfaches Programmbeispiel, das zwei eigene Funktionsobjekte `myequal` und `addvalue` definiert und zwei von der Standardbibliothek vordefinierte Funktionsobjekte (`less` und `logical_not()`) verwendet.

```
// stl4.cpp
#include <iostream>
#include <functional>
using namespace std;

// Eigenes Funktionsobjekt myequal
template <class T>
```

```
class myequal : public binary_function<T,T,bool> {
   public:
   bool operator() (const T& a, const T& b) const {
      return a==b;
   }
};

// Eigenes Funktionsobjekt addvalue
template <class T>
class addvalue : public unary_function<T,T> {
private:
   T val;
public:
   addvalue(const T& newval) : val(newval) { };
   T operator() (const T& arg) const {
      return arg+val;
   }
};

int main() {
   int i;
   cout << "Bitte einen Integer eingeben: ";
   cin >> i;
   // Vordefiniertes Funktionsobjekt
   if( logical_not<int>()(i) ) {
      cout << "Sie haben 0 eingegeben\n";
   }
   // Eigenes Funktionsobjekt
   if ( myequal<int>()(i,1) ) {
      cout << "Es wurde 1 eingegeben" << endl;
   }
   // Eigenes Funktionsobjekt
   cout << addvalue<int>(5)(i) << endl;
   // Vordefiniertes Funktionsobjekt
   if( less<int>() (i, 100) ) {
      cout << "i ist kleiner als 100\n";
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Bitte einen Integer eingeben: 0
Sie haben 0 eingegeben
5
i ist kleiner als 100
```

```
Bitte einen Integer eingeben: 1
Es wurde 1 eingegeben
6
i ist kleiner als 100

Bitte einen Integer eingeben: 100
105
```

Bei genauerem Hinsehen dürfte sich hier der ein oder andere an Funktionszeiger erinnert fühlen. Im Grunde sind die Funktionsobjekte ähnlich wie Funktionszeiger, nur erheblich mächtiger. Neben den hier gezeigten vordefinierten Funktionsobjekten `less` und `logical_not` gibt es noch folgende vordefinierte Funktionsobjekte:

Arithmetik-Funktor	Operation
`plus<T>`	`x+y`
`minus<T>`	`x-y`
`multiplies<T>`	`x*y`
`divides<T>`	`x/y`
`modulus<T>`	`x%y`
`negate<T>`	`-x`
Vergleichs-Funktor	**Operation**
`equal_to<T>`	`x==y`
`not_equal_to<T>`	`x!=y`
`less<T>`	`x<y`
`less_equal<T>`	`x<=y`
`greater<T>`	`x>y`
`greater_equal<T>`	`x>=y`
Logik-Funktor	**Operation**
`logical_and<T>`	`x&&y`
`logical_or<T>`	`x||y`
`logical_not<T>`	`!x`

Tabelle 5.2 Vordefinierte Funktionsobjekte der Standardbibliothek

Funktionsadapter

Funktionsadapter sind nichts anderes als Funktionsobjekte, die mit anderen Funktionsobjekten kooperieren, um diese wiederum an andere Anforderungen anzupassen (verschachtelte Funktionsobjekte, wenn Sie so wollen). Sinn und Zweck eines Funktionsadapters ist es, möglichst mit den vorhandenen Funktoren auszukommen, um sich das Schreiben neuer Funktoren zu ersparen.

not1, not2

Der Funktionsadapter `not1` ist für unäre Funktionsobjekte gedacht, die einen booleschen Wert zurückgeben. `not1` entspricht dabei einer logischen Negierung (`!ausdruck`).

`not2` ist gleichwertig zu `not1`, nur dass dieser Adapter für binäre Funktionsobjekte verwendet wird.

bind1st, bind2nd

Die Adapter `bind1st` bzw. `bind2nd` erzeugen aus einem binären Funktionsobjekt ein unäres, indem das erste Argument (`bind1st`) oder das zweite Argument (`bind2nd`) mit einem festen Wert belegt wird. Zum Beispiel können Sie das selbstgeschriebene Funktionsobjekt `addvalue` aus dem Listing *stl4.cpp* einfacher mit `bind2nd` umschreiben.

ptr_fun

Mit dieser Funktion können Sie einen Zeiger auf eine Funktion in einen Funktor umwandeln. Sie hat als Argument einen Zeiger, der einen (unär) oder zwei (binär) Parameter haben kann. Als Rückgabewert erhalten Sie aus diesem Funktionsadapter ein Funktionsobjekt, das wie eine Funktion aufgerufen werden kann.

Hierzu ein Listing, das die beschriebenen Funktionsadapter im Einsatz zeigen soll:

```
// stl5.cpp
#include <iostream>
#include <functional>
#include <cmath>
using namespace std;

template <class T>
class myequal : public binary_function<T,T,bool> {
public:


   bool operator() (const T& a, const T& b) const {
      return a==b;
   }
};

// Quadratwurzel-Funktion
double (*func)(double) = sqrt;

int main() {
```

```
   int i;
   cout << "Bitte einen Integer eingeben: ";
   cin >> i;
   if( not1(logical_not<int>())(i) ) {
      cout << "Sie haben nicht 0 eingegeben\n";
   }
   // Eigenes Funktionsobjekt
   if ( not2(myequal<int>())(i,1) ) {
      cout << "Es wurde nicht 1 eingegeben" << endl;
   }
   // Alternativ für addvalue
   cout << bind2nd( plus<int>(), 5)(i) << endl;

   // Aufruf als Funktion
   cout << func(1.1) << "\n";
   // Aufruf als Funktor
   cout << ptr_fun(func)(1.1) << "\n";
   return 0;
}
```

Das Programm bei der Ausführung:

```
Bitte einen Integer eingeben: 1
Sie haben nicht 0 eingegeben
6
1.04881
1.04881

Bitte einen Integer eingeben: 0
Es wurde nicht 1 eingegeben
5
1.04881
1.04881
```

5.3.3 Allokator

Da auf den folgenden Seiten häufiger von Allokatoren die Rede sein wird, hier kurz die Definition. Der Name sagt im Prinzip schon, worum es sich handelt: Ein Allokator ist verantwortlich für die Speicherbeschaffung von Containern. Gewöhnlich werden solche Allokatoren vom System bereitgestellt, und man muss sich nicht darum kümmern; es ist aber auch möglich, eigene Allokatoren zu schreiben. Mit ihnen lässt sich beispielsweise eine Speicherbereinigung (*Garbage Collection*) schreiben, wie dies in Java üblich ist. Mehr zu den Allokatoren erfahren Sie in Abschnitt 5.3.7, »Allokatoren«.

5.3.4 Iteratoren

Wie Sie bereits gelesen haben, werden Iteratoren von Algorithmen benutzt, um Container zu durchlaufen. Im Prinzip wirken Iteratoren wie gewöhnliche Zeiger. In diesem Abschnitt soll detaillierter auf diese Iteratoren und deren Eigenschaften eingegangen werden.

Hinweis [«]

Bei der Beschreibung der Iteratoren handelt es sich zunächst um eine theoretische Einführung. Die Praxis werden Sie direkt im Einsatz mit den Containern bzw. Algorithmen kennenlernen.

Zustände von Iteratoren

Wie ein gewöhnlicher Zeiger kann auch ein Iterator verschiedene Zustände haben:

- Wenn ein Iterator erzeugt wird, kann dieser nicht gleich mit einem Container verbunden werden. Eine solche Verbindung kann erst nachträglich gemacht werden. Natürlich ist ein Iterator nicht dereferenzierbar und vergleichbar mit einem Zeiger auf `0`.
- Wenn ein Iterator mit einem Container verbunden wird, zeigt dieser (sofern nicht anders angegeben) standardmäßig auf den Anfang des Containers. Diese Position kann mit der Methode `begin()` ermittelt werden. Jetzt kann man mit dem Iterator auf die einzelnen Elemente des Containers zugreifen. Die Operationen, die mit einem Iterator nun durchgeführt werden können, entsprechen denen der Zeiger (siehe Tabelle 5.3). Diese Operationen sind allerdings noch abhängig vom Typ des Iterators. Die letzte Position eines Iterators ist die `end()`-Position, sie ist nicht dereferenzierbar.

Operation	Bedeutung	Erlaubt für
`++pos`, `pos++`	nächste Position vorwärts	alle Iteratoren
`--pos`, `pos--`	nächste Position rückwärts	bidirektionale und Random-Access-Iteratoren
`*pos`	Zugriff auf das Objekt	alle Iteratoren
`pos->`	Elementzugriff	alle Iteratoren
`pos[n]`	gleich zu `*(pos + n)`	Random-Access-Iteratoren
`pos1-pos2`	Anzahl Objekte zwischen `pos1` und `pos2`	alle Iteratoren
`(pos+n)`	Position `n` hinter `pos`	Random-Access-Iteratoren
`(pos-n)`	Position `n` vor `pos`	Random-Access-Iteratoren

Tabelle 5.3 Operationen, die auf Iteratoren durchgeführt werden können

Operation	Bedeutung	Erlaubt für
==, !=	Gleichheit, Ungleichheit	alle Iteratoren
=	Zuweisung	alle Iteratoren
+=, -=	zusammengesetzte Zuweisung	Random-Access-Iteratoren
>, >=, <, <=	kleiner/größer als ...	Random-Access-Iteratoren

Tabelle 5.3 Operationen, die auf Iteratoren durchgeführt werden können (Forts.)

Kategorien von Iteratoren

In Tabelle 5.3 haben Sie bereits gesehen, dass STL verschiedene Iteratoren anbietet. Für die jeweiligen Container liefert STL unterschiedliche Iteratoren, die in fünf Kategorien eingeteilt werden. Diese Kategorien entsprechen den Fähigkeiten der Iteratoren.

- *Input-Iterator* (`input_iterator`) – dieser Iterator wird nur lesend auf Objekte verwendet und kann nur vorwärts bewegt werden. Input-Iteratoren sind lediglich für einen Durchlauf geeignet, da es wegen der Stream-Eigenschaft nicht möglich ist, sich einen speziellen Iterator-Wert zu merken (lesender Zugriff).
- *Output-Iterator* (`output_iterator`) – dieser Iterator ist das Gegenstück zum Input-Iterator und kann nur zum schreibenden Zugriff in der Vorwärtsbewegung auf Objekte verwendet werden. Auch dieser Iterator ist lediglich für einen Durchlauf geeignet. Außerdem sollte möglichst nur ein Output-Iterator auf einem Container aktiv sein.
- *Forward-Iterator* (`forward_iterator`) – dieser Iterator kann wie der Input- und der Output-Iterator nur in der Vorwärtsbewegung verwendet werden. Allerdings kann er lesend und schreibend auf Objekte angewendet werden, so dass es auch möglich ist, Werte eines Iterators zwischenzuspeichern, um das Element später wiederzufinden. Außerdem ist mit diesem Iterator ein mehrfaches Durchlaufen in einer Richtung möglich. Ein solches Beispiel ist mit verketteten Listen gegeben.
- *Bidirektionaler Iterator* (`bidirectional_iterator`) – dieser Iterator kann alles, was auch der Forward-Iterator kann, nur dass er auch noch zusätzlich rückwärts bewegt werden kann – was beispielsweise bei doppelt verketteten Listen gegeben sein muss.
- *Random-Access-Iterator* (`random_access_iterator`) – dieser Iterator ist wiederum eine Weiterbildung des bidirektionalen Iterators, und er kann alles, was dieser kann, nur ist zusätzlich noch ein wahlfreier Zugriff auf Objekte möglich. Dieser wahlfreie Zugriff wird zum Beispiel bei Vektoren durch den Indexoperator `operator[]` realisiert. Des Weiteren lassen sich damit auch arithmetische Operationen durchführen (siehe auch Tabelle 5.3).

Es gibt außerdem noch einen reversen Iterator (`reverse_iterator`), bei dem die Bedeutungen von `++` und `--` vertauscht sind. Ein solcher *Reverse-Iterator* ist bei einem bidirektionalen Iterator immer möglich. Dieser durchläuft praktisch einen Container mit ++-Operation rückwärts. Der Beginn und das Ende eines solchen Containers werden durch `rbegin()` und `rend()` markiert – wobei `rbegin()` auf das letzte Element verweist und `rend() -1` auf das erste. Einige Container stellen diese Reverse-Iteratoren zur Verfügung.

Daneben ist außerdem auch ein Typ `const_iterator` für Iteratoren definiert. Dieser wird verwendet, wenn ein Container konstant sein muss, denn dann muss logischerweise auch der Iterator konstant sein. Man kann schließlich nicht eine Funktion für die Ausgabe von Elementen eines Containers schreiben, wenn als Parameter eine Referenz auf einen konstanten Container übergeben wird. Für solche Zwecke stellt STL auch einen konstanten Iterator zur Verfügung, mit dem nur lesende Zugriffe möglich sind.

Für alle Container-Klassen (abgesehen von den Adapter-Klassen) ist auf jeden Fall der Datentyp `iterator` zur Darstellung und Verwendung von Iteratoren definiert. Dieser Iterator-Typ ist auch der am meisten verwendete. Beispielsweise wird mit der Deklaration

```
vector<float>::iterator position;
```

ein Iterator `position` angelegt, der einen `float`-Vektor (Array) durchlaufen und auf die einzelnen Elemente im Vektor zugreifen kann.

Einfüge-Iteratoren

Wenn Sie mit dem gewöhnlichen Iterator ein Element im Container einfügen würden, würden Sie das aktuelle Element, auf das der Iterator verweist, überschreiben. Zum Einfügen neuer Elemente in einem Container gibt es sogenannte *Insert-Iteratoren*. STL stellt drei verschiedene Insert-Iteratoren zur Verfügung: `front_inserter`, `back_inserter` und `inserter`. Natürlich lassen sich diese Insert-Iteratoren nicht willkürlich verwenden. So ist es logischerweise nicht möglich, einen `front_inserter`-Iterator auf einen Vektor anzuwenden, weil es nicht erlaubt ist, ein Element am Container-Anfang einzufügen.

Mit `front_inserter(container)` fügen Sie das einzufügende Element am Anfang des Containers ein. `back_inserter(container)`, das Gegenstück zu `front_inserter`, fügt das neue Element am Ende des Containers ein. Mit dem allgemeinen `inserter(container, pos)` können Sie Elemente an einer beliebigen Position im Container einfügen.

Stream-Iterator

Die *Stream-Iteratoren* werden dazu verwendet, mit Ein- bzw. Ausgabeströmen zu arbeiten. Dabei wird zwischen zwei Stream-Iteratoren unterschieden: `ostream_iterator` und `istream_iterator`. Um die Stream-Iteratoren zu verwenden, müssen Sie außerdem die Header-Datei *<fstream>* einfügen.

Die beiden Stream-Iteratoren werden gewöhnlich dazu verwendet, von Strömen mit den bekannten Operatoren `operator<<` und `operator>>` sequentiell zu lesen bzw. zu schreiben. Dabei ist der `ostream_iterator` der Ausgabe-Iterator und der `istream_iterator` der Eingabe-Iterator. Ein einfaches Beispiel dazu:

```
// Anzeige aller durch Zwischenraumzeichen
// getrennten Zeichenfolgen:
ifstream source("Dateiname");
istream_iterator<string> pos(source), end;
while(pos != end) {
  cout << *pos  << endl;
  ++pos;
}
```

Mit der Dereferenzierung von `pos` in der Schleife können Sie den gelesenen Wert ausgeben lassen. Wenn Sie diesen Codeausschnitt in der Praxis testen, werden Sie feststellen, dass der Iterator `istream_iterator` alle bekannten Whitespaces wie Leerzeichen, Tabulatorzeichen und Zeilenendezeichen vor einem Element ignoriert und zwischen zwei Elementen als Trennzeichen verwendet wird.

Das Gegenstück dazu ist der `ostream_iterator`, der zum Schreiben von Elementen verwendet wird. Allerdings muss man beachten, dass der `ostream_iterator` aufeinanderfolgende Elemente mit dem Operator `<<` gewöhnlich ohne Trennzeichen nacheinander in den Stream schiebt. Hierbei ist es möglich, diesem Iterator bei der Definition eine Zeichenkette vom Typ `char*` mitzugeben, die nach jedem Element als Trennzeichen fungieren soll:

```
ifstream Quelle("C:/Dev-Cpp/test.cpp");
ofstream Ziel("C:/Dev-Cpp/test.bak");
istream_iterator<string> iPos(Quelle), Ende;
ostream_iterator<string> oPos(Ziel, "\n");

while(iPos != Ende)
   *oPos++ = *iPos++;
```

Hier wurde nach jedem Trennzeichen ein Newline-Zeichen angehängt, so dass jeder neue String in eine neue Zeile der Datei geschrieben wird.

Hinweis [«]

An dieser Stelle sei noch angemerkt, dass die String- und Stream-Klassen noch gar nicht behandelt wurden und erst ab Kapitel 7, »C++-Standardbibliothek«, folgen.

Distanzen (Iterator-Funktionen)

Die Operationen + und – sind nur mit den Random-Access-Iteratoren erlaubt. Daher wurde die Funktion `distance()` eingeführt, um die Anzahl der Elemente in einem Bereich zu bestimmen, der zwischen zwei Input-Iteratoren definiert ist. Dies wird häufig verwendet, um zu ermitteln, an welcher Position sich ein Element im Container befindet. Man kann für den ersten Parameter den Iterator `begin()` verwenden:

```
// Iteratoren initialisieren
vector<int>::iterator Pos1 = ...;
vector<int>::iterator Pos2 = ...;
// Anzahl der Elemente im Bereich zwischen
// Pos1 und Pos2 ausgeben
cout << distance(Pos1, Pos2) << endl;
```

Eine weitere Funktion wurde mit `advance()` eingeführt, mit der ein Input-Iterator um `n` Positionen vorwärts geschoben werden kann. Wenn es sich bei dem Iterator um einen bidirektionalen oder Random-Access-Iterator handelt, kann man auch negative Werte verwenden, um den Iterator rückwärts zu verschieben:

```
// Listen-Iterator (bidirektional!)
list<Daten>::iterator aktPos;
// Um 10 Positionen vorwärts
advance(aktPos, 10);
// Um drei Positionen rückwärts
advance(aktPos, -3);
```

Beide Funktionen verwenden intern + und – für Random-Access-Iteratoren oder in anderen Fällen ++ bzw. `--`. Die Syntax der beiden Funktionen sieht wie folgt aus:

```
// schaltet i um n-Positionen vor bzw. zurück, falls n < 0
template <class InputIterator, class Distance>
void advance(InputIterator& i, Distance n);
// Gibt den Abstand zwischen zwei Iteratoren zurück.
// Dabei muss last von first aus erreichbar sein.
template <class InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

5.3.5 Container

In Abschnitt 5.3.1, »Konzept von STL«, wurde bereits beschrieben, was Container sind und vor allem, wofür sie verwendet werden. Bevor auf den folgenden Seiten die einzelnen Container, die STL anbietet, erläutert werden, soll hier noch auf die gemeinsamen Methoden und Datentypen eingegangen werden, die alle Container besitzen.

[»] **Hinweis**

Die Praxisbeispiele hierzu erfolgen dann in den entsprechenden Container-Abschnitten.

Datentypen

Damit die Schnittstelle zum Container bei einem Programm zur Kompilierzeit immer gleich ist, bieten diese auch »eigene« Datentypen an. Mit dem `iterator` haben Sie bereits einen solchen Datentyp verwendet, zum Beispiel:

```
vector<int>::iterator bezeichner;
```

Diese Verwendung kann identisch sein mit einem Zeiger wie `int*`. In der folgenden Tabelle finden Sie die zur Verfügung gestellten Datentypen aller Container der STL. In der Tabelle stellt dabei `C` den Datentyp des Containers dar (beispielsweise `stack<int>`, `vector<double>` etc.).

Datentyp	Bedeutung
`C::value_type`	Datentyp des Container-Elements (beispielsweise ist `vector<int>::value_typ` gleichwertig zu `int`)
`C::reference`	Referenz auf ein Container-Element
`C::const_reference`	Referenz auf ein Container-Element, das nicht verändert werden kann
`C::iterator`	Typ des Iterators
`C::const_iterator`	Typ des Iterators, der nicht zum Manipulieren von Daten verwendet werden kann
`C::difference_type`	integraler Typ mit Vorzeichen
`C::size_type`	integraler Typ ohne Vorzeichen für Größenangaben
`C::reverse_iterator`	Typ des reversen (rückwärtslaufenden) Iterators
`C::const_reverse_iterator`	Typ des reversen (rückwärtslaufenden) Iterators, der nicht zum Manipulieren von Daten verwendet werden kann

Tabelle 5.4 Datentypen für Container

Methoden

Alle Container bieten die in der folgenden Tabelle gezeigten Methoden an, weshalb bei der Beschreibung der Container selbst nicht mehr darauf eingegangen wird. Die Methoden `begin()` und `end()` haben Sie bereits in der Praxis gesehen. In der Tabelle wird wieder `C` für die Bezeichnung des Container-Typs verwendet.

Methode	Beschreibung
`C()`	Default-Konstruktor, der einen leeren Container erzeugt
`C(const C&)`	Kopierkonstruktor
`~C()`	Destruktor
`iterator begin()`	Anfang eines Containers
`iterator end()`	eine Position nach dem letzten Container
`const_iterator begin()` `const_iterator end()`	wie eben, allerdings nur lesend anwendbar
`iterator rbegin()`	Reversible Container – verweist auf das letzte Element.
`iterator rend()`	Reversible Container – verweist auf fiktive Position vor dem ersten Element.
`size_type max_size()`	maximal erlaubte Größe eines Containers
`size_type size()`	aktuelle Größe eines Containers
`bool empty()`	Gibt `true` zurück, wenn Container leer ist. Gleichwertig zu `size() == 0` oder `begin()==end()`.
`void swap(C&)`	zwei Container vertauschen
`C& operator=(const C&)`	Zuweisungsoperator

Tabelle 5.5 Methoden für alle Container

Neben den hier genannten Methoden gibt es noch die relationalen Operatoren ==, !=, <, >, <= und >=. Die Operatoren == und =! werden zum Vergleich zweier Container-Größen und zum Vergleich der Elemente `T` (wenn `operator==()` definiert ist) verwendet. Alle anderen Operatoren basieren auf einem lexikographischen Vergleich (`operator<()` muss als Ordnungsrelation definiert sein, siehe Tabelle 5.6).

Operator	Beschreibung
`bool operator==(const C&)`	Operator ==
`bool operator!=(const C&)`	Operator !=
`bool operator<(const C&)`	Operator <
`bool operator>(const C&)`	Operator >
`bool operator<=(const C&)`	Operator <=
`bool operator>=(const C&)`	Operator >=

Tabelle 5.6 Relationale Operatoren für Container

Sequentielle Container

Sequentielle Container sind Container, deren Elemente linear angeordnet sind. In STL sind dies drei verschiedene Arten von Containern, nämlich `vector`, `list` und `deque`, die im Folgenden in der Praxis erläutert werden.

Ein Hauptkriterium dafür, welchen Container man einsetzt, dürfte das gewünschte Laufzeitverhalten sein. Eine Liste (`list`) wird daher in der Praxis immer dann eingesetzt, wenn viele neue Elemente eingefügt bzw. gelöscht werden sollen. Werden die Elemente vorwiegend am Ende eingefügt, kann man ein Array (`vector`) verwenden. Sollen die neuen Elemente häufig an einem der beiden Enden eingefügt werden, können auch die Warteschlangen mit zwei Enden (`deque`) vorn und hinten verwendet werden.

So ist das Einfügen von Elementen in einen Vektor irgendwo in der Mitte relativ langsam, da Vektor-Elemente dahinter verschoben werden müssen. Auch `deque` ist im Vergleich zur Liste langsamer, da hierbei zwei Zeiger verändert werden müssen. `list` hingegeben muss für solche Zwecke nur einen Zeiger verändern und ist für das Einfügen in der Mitte am besten geeignet. Somit hängt die Auswahl des Containers stark von der Anzahl der Elemente ab, die eingefügt werden müssen. Vorüberlegungen sind hier also Pflicht.

Laufzeitklassen

Die Laufzeiten von Algorithmen werden in bestimmte Klassen aufgeteilt. In der Tabelle 5.7 finden Sie eine solche Einteilung der Laufzeitklassen. `N` steht hierbei für die Anzahl der Elemente.

Laufzeit	Bezeichnung	Beispiel
O(1)	konstant	Die Operationen `push` und `pop` bei Stacks benötigen im günstigsten Fall konstante Zeit. Sie sind damit unabhängig von der Elementanzahl. Dasselbe gilt für den Zugriff auf ein Vektor-Element mit dem `[]`-Operator.
O(log N)	logarithmisch	Die binäre Suche oder das Suchen in höhenbalancierten Bäumen fällt in diese Kategorie. Bei linearer Erhöhung der Elementanzahl steigt die benötigte Suchzeit immer langsamer.
O(N)	linear	Die sequentielle Suche besitzt ein lineares Laufzeitverhalten. Die benötigte Zeit steigt proportional zur Elementanzahl.
O(N*log N)	N*log N	Ein optimales, auf Schlüsselvergleichen basierendes Sortierverfahren kann bestenfalls O(N * log N)-Zeit besitzen.
$O(N^2)$	quadratisch	Das Sortierverfahren Bubblesort hat ein quadratisches Laufzeitverhalten.

Tabelle 5.7 Die verschiedenen Laufzeitklassen

Laufzeit	Bezeichnung	Beispiel
$O(2^N)$	exponentiell	Beispielsweise alle Probleme, die in die Kategorie der NP-Vollständigkeit fallen, benötigen zu ihrer Lösung exponentielle Laufzeit.

Tabelle 5.7 Die verschiedenen Laufzeitklassen (Forts.)

Worst case, best case und average case

Häufig ist auch bei den Laufzeiten von Algorithmen die Rede vom *worst case*, *best case* und *average case*. Der schlechteste Fall, der bei einem Algorithmus auftreten kann, wird als *worst case* bezeichnet. Der beste Fall wird als *best case* und die durchschnittliche Laufzeit eines Algorithmus als *average case* bezeichnet. Zwar wird in der Praxis immer vom schlechtesten Fall (*worst case*), der eintreten kann, ausgegangen, aber man sollte dennoch auch *best case* und *average case* beachten.

Der beste Fall (*best case*) ist zum Beispiel gegeben, wenn bei einer sequentiellen Suche das Element als Erstes in einer Datenstruktur steht. In diesem Fall benötigt die Suche eine Laufzeit von 1. Der beste Fall wird auch als Omega-Notation ausgedrückt: $\Omega(1)$.

Sollte der Fall auftreten, dass das Laufzeitverhalten für den besten und schlechtesten Fall identisch ist, wird die Theta-Notation verwendet: Θ-Notation. Dies kann beispielsweise bei einem Stack gegeben sein, wenn die Operationen `push` und `pop` im besten und schlimmsten Fall jeweils eine konstante Laufzeit ergeben. Dann gilt für das Laufzeitverhalten $\Theta(1)$.

Aber auch der durchschnittliche Fall ist für die Praxis interessant. Ein häufig verwendeter Algorithmus ist Quicksort, der schlimmstenfalls $O(N^2)$ an Zeit benötigen kann. Dies ist der ungünstigste Fall, aber in der Praxis hat sich gezeigt, dass Quicksort meistens im durchschnittlichen Fall eine Zeit von N*log N benötigt – weshalb dieser Algorithmus auch gerne eingesetzt wird.

Hinweis [«]

Wie gut oder schlecht eine Lösung ist, hängt immer von der Problemstellung ab.

Die Container-Klasse »vector«

Hinweis [«]

Vektoren sind hier nicht als mathematische Vektoren zu verstehen, sondern als eine Art Array, das sich nach Bedarf automatisch vergrößert (und verkleinert).

Vektoren (`vector`) werden in der Praxis verwendet, wenn der Zugriff auf ein Element sehr schnell sein soll – O(1) –, was hier über die Indexnummer sehr effizi-

ent und schnell geschieht (über Adressen). Der Nachteil ist, dass das Einfügen im Inneren eines Vektors relativ langsam ist – O(N) –, weil alle Elemente dahinter um eine Stelle verschoben werden müssen.

Ein `vector` wird mit der Header-Datei <*vector*> eingebunden. Neben den Datentypen der Tabelle 5.4 bietet ein Vektor noch folgende weitere öffentliche Datentypen:

Datentyp	Bedeutung
`pointer`	Zeiger auf Vektor-Element
`const_pointer`	Zeiger auf Vektor-Element (nur lesend anwendbar)

Tabelle 5.8 Öffentliche Datentypen für »vector«

Die Deklaration der Container-Klasse hat folgende Syntax:

```
template<class T> class vector;
```

Neben den allgemeinen Methoden (siehe Tabelle 5.5), die alle Container-Klassen enthalten, bietet die Container-Klasse `vector` noch weitere zusätzliche Methoden an, die alle mit dieser Container-Klasse verwendet werden können. Die folgenden Tabellen (5.9 bis 5.13) sind nach Themen sortiert. `T` steht wieder für den Typ.

Methode	Bedeutung
`T& operator[](size_type n);`	Gibt eine Referenz auf das `n`. Element des Vektors zurück (beispielsweise `T[n]`; ist `n>=size()`, dann ist das Verhalten undefiniert).
`T& at(size_type n);`	Wie `operator[]`; nur wenn `n>=size()`, dann wird eine Exception vom Typ `out_of_range` ausgeworfen.
`T& front();`	Liefert eine Referenz auf das erste Element im Vektor zurück.
`T& back();`	Liefert eine Referenz auf das letzte Element im Vektor zurück.

Tabelle 5.9 Methoden für den Elementzugriff

[»] Hinweis

Alle Methoden für den Elementzugriff bieten auch eine `const`-Version an, mit der nur noch lesend auf die Elemente zugegriffen werden darf.

Methode	Bedeutung
`size_type capacity() const;`	Maximal verfügbare Größe der im Vektor zu speichernden Elemente, ohne dass erneut Speicher alloziert werden muss.
`void reserve(size_type n);`	Vergrößerung des Wertes von `capacity()`
`void resize( size_type n, t = T() );`	Ändert die Container-Größe. Hierbei werden `n-size()` `t`-Elemente am Ende hinzugefügt bzw. `size()-n` Elemente gelöscht (abhängig davon, ob `n` größer oder kleiner als die aktuelle Größe ist).

Tabelle 5.10 Größenbezogene Methoden

Methode	Bedeutung
`void push_back( const T& obj );`	Fügt `obj` hinter dem letzten Element im Vektor ein.
`iterator insert( iterator pos, const T& obj );`	Fügt das Objekt `obj` an der Position `pos` des Vektors ein. Alle Elemente dahinter werden um eine Position nach hinten verschoben.
`void insert( iterator pos, size_type n, const T&obj );`	Fügt `n` Kopien vom Objekt `obj` an der Position `pos` des Vektors ein. Alle Elemente dahinter werden um `n` Positionen verschoben.
`template <class InputIterator> void insert( iterator pos, InputIterator first, InputIterator last );`	Fügt die Elemente im Bereich `[first, last)` vor der Position `pos` ein. `first` und `last` zeigen nicht auf den Container, für den `insert` aufgerufen wird.

Tabelle 5.11 Methoden zum Einfügen neuer Elemente

Methode	Bedeutung
`void pop_back();`	Letztes Element im Vektor löschen.
`iterator erase(iterator pos);`	Löscht das Element mit der Position `pos`. Der zurückgegebene Iterator verweist auf den Nachfolger von `pos` oder auf `end()`, falls das letzte Element gelöscht wurde. Somit werden alle Elemente hinter dem zu löschenden um eine Position nach vorn »gezogen«.
`iterator erase( iterator first, iterator last );`	Löscht die Elemente im Bereich `[first,last)`. Der zurückgegebene Iterator verweist auf den Nachfolger von `last` oder auf `end()`, falls `last` nicht existiert. Alle Elemente vor dem zu löschenden werden somit um `last-first` Positionen nach vorn gezogen.
`void clear();`	Löscht alle Vektor-Elemente, so dass der Vektor die Länge 0 besitzt.

Tabelle 5.12 Methoden zum Löschen von Elementen

Methode	Bedeutung
`vector<T, Allocator>& operator=( const vector<T, Allocator>& vect );`	Zuweisung an den aktuellen Vektor (`*this`) mit dem kompletten Inhalt von `vect` (beispielsweise `myvect = vect`)
`template <class InputIterator> void assign(InputIterator first, InputIterator last);`	Container löschen und anschließend die Elemente aus dem Iterator-Bereich `[first,last)` einfügen.
`void assign( size_type n, const T& v );`	Container löschen und gegen `n` Elemente von `v` ersetzen.
`void swap( vector<T, Allocator>& vect );`	Inhalte von zwei Vektoren vertauschen.

Tabelle 5.13 Zuweisen und Tauschen von Vektoren

Die Verwendung der einzelnen Methoden ist ähnlich, wie die von Methoden einer Klasse auch. Es empfiehlt sich, mit den einzelnen Methoden zu experimentieren, um sich damit vertraut zu machen. Natürlich keine Theorie ohne Praxis. Daher folgt hierzu ein kleines Listing, das einige dieser Methoden verwendet. Dabei werden ein `int`-Vektor und ein Vektor einer Klasse mit dem Namen `A` verwendet.

```
// stl_vector.cpp
#include <vector>
#include <iostream>
using namespace std;

class A {
private:
   int ival;
   float fval;
public:
   // Konstruktor
   A( int i=0, float f=0.0f ): ival(i), fval(f) {}
   // Destruktor
   ~A( ) {}
   void anzeigen() const {
      cout << ival << " : " << fval << "\n";
   }
};

int main() {
   // ein int-Vektor mit 5 Elementen
   vector<int> Vectorint(5);
   // ein Vektor der Klasse A mit 3 Elementen
   vector<A> VectorA(3);
```

```
// int-Vektor mit Werten füllen
for(size_t i = 0; i < Vectorint.size(); ++i)
   Vectorint[i] = i*i;
// A-vector mit Werten füllen
float f=1.1;
for(size_t i=0; i < VectorA.size(); ++i) {
   VectorA[i] =  A(i, f);
   f*=2;
}
// int-Vektor dynamisch vergrößern (Zahl 1111 anhängen)
Vectorint.insert(Vectorint.end(), 1111);
// A-Vektor um ein Element
// erweitern (111, 111.111 anhängen)
VectorA.insert(VectorA.end(), A(111, 111.111));
// ... hinten anhängen geht auch so ...
VectorA.push_back( A(222, 222.222));
// Einzelne Elemente ausgeben über
// ... die Benutzung als Array
cout << "int-Vektor: \n";
for(size_t i = 0; i < Vectorint.size(); ++i)
   cout << Vectorint[i] << ", ";
cout << endl << "A-Vektor: \n";
for(size_t i = 0; i < VectorA.size(); ++i)
   VectorA[i].anzeigen();
// Zugriff auf einzelne Elemente über den Iterator
cout << "\nint-Vektor (mit Iterator): \n";
for(vector<int>::const_iterator myiter=Vectorint.begin();
    myiter != Vectorint.end(); ++myiter )
   cout << *myiter << endl;
cout << "A-Vektor (mit Iterator): \n";
for( vector<A>::const_iterator myiter = VectorA.begin();
     myiter != VectorA.end(); ++myiter )
   myiter->anzeigen();
cout << endl;
// neuen int-Vektor mit Vectorint.size()
// Elemente erzeugen
// alle int-Werte mit 0 vorbelegen
vector<int> Vectorint_Neu(Vectorint.size(), 0);
// alle A-Werte mit 0 und 0.0f vorbelegen
vector<A> VectorA_Neu(VectorA.size(), A(0, 0.0f));

// Elemente mit swap() tauschen (schnelle Methode)
Vectorint_Neu.swap(Vectorint);
VectorA_Neu.swap(VectorA);
```

```
   cout << "\nNach den swap-Aktionen\n";
   cout << "Vectorint_Neu: \n";
   for(size_t i = 0; i < Vectorint_Neu.size(); ++i)
      cout << Vectorint_Neu[i] << ", ";
   cout << endl;
   cout << "Vectorint: \n";
   for(size_t i = 0; i < Vectorint.size(); ++i)
      cout << Vectorint[i] << ", ";
   cout << endl;
   cout << "VectorA_Neu: \n";
   for(size_t i = 0; i < VectorA_Neu.size(); ++i)
      VectorA_Neu[i].anzeigen();
   cout << endl;
   cout << "VectorA: \n";
   for(size_t i = 0; i < VectorA.size(); ++i)
      VectorA[i].anzeigen();
   cout << endl;

   // Alle Elemente löschen
   Vectorint.clear();
   if(Vectorint.empty())
      cout << "Vectorint ist leer\n";
   else
      cout << "Vectorint ist nicht leer\n";

   // Löschen von einzelnen Elementen über den Iterator
   for( vector<int>::iterator myiter1=Vectorint_Neu.begin();
        myiter1 != Vectorint_Neu.end();
        ++myiter1 ) {
      if( *myiter1 == 9 ) {
         // Element mit Inhalt 9 bis zum Ende löschen
         Vectorint_Neu.erase(myiter1, Vectorint_Neu.end() );
         break;
      }
   }
   cout << "Vectorint_Neu: \n";
   for(size_t i = 0; i < Vectorint_Neu.size(); ++i)
      cout << Vectorint_Neu[i] << ", ";
   cout << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
int-Vektor:
0, 1, 4, 9, 16, 1111,
```

```
A-Vektor:
0 : 1.1
1 : 2.2
2 : 4.4
111 : 111.111
222 : 222.222

int-Vektor (mit Iterator):
0
1
4
9
16
1111
A-Vektor (mit Iterator):
0 : 1.1
1 : 2.2
2 : 4.4
111 : 111.111
222 : 222.222

Nach den swap-Aktionen
Vectorint_Neu:
0, 1, 4, 9, 16, 1111,
Vectorint:
0, 0, 0, 0, 0, 0,
VectorA_Neu:
0 : 1.1
1 : 2.2
2 : 4.4
111 : 111.111
222 : 222.222

VectorA:
0 : 0
0 : 0
0 : 0
0 : 0
0 : 0

Vectorint ist leer
Vectorint_Neu:
0, 1, 4,
```

Die Container-Klasse »list«

Die Container-Klasse `list` ist eine fast äquivalente Klasse für lineare Listen, die intern durch eine Verkettung dargestellt wird. Allerdings ist, im Gegensatz zur Container-Klasse `vector`, das Einfügen im Inneren erheblich schneller – O(1) –, da hier nur die Adressen von Zeigern verändert werden müssen und nicht ganze Elemente verschoben werden. Viele bekannte Methoden, die Sie im Abschnitt über die `vector`-Container-Klasse kennengelernt haben, finden Sie auch wieder bei der Container-Klasse `list`. Selbstverständlich gibt es hier auch noch einige spezielle Listenoperationen.

Die Liste von STL ist allerdings immer eine doppelt verkettete Liste, die das Hinzufügen und Wegnehmen von Elementen sowohl am Anfang als auch am Ende der Liste erlaubt. Wenn Sie eine Liste vom Typ `list` erzeugen und verwenden wollen, müssen Sie die Header-Datei *<list>* mit einbinden, womit Ihnen alle Typen und die hier noch folgenden Methoden zur Verfügung stehen. Neben den Datentypen aus Tabelle 5.4 bietet eine Liste noch folgende weitere öffentliche Datentypen:

Datentyp	Bedeutung
`pointer`	Zeiger auf Listen-Element
`const_pointer`	Zeiger auf Listen-Element (nur lesend anwendbar)

Tabelle 5.14 Öffentliche Datentypen für »list«

Die Deklaration der Container-Klasse `list` hat folgende Syntax:

```
template<class T> class list;
```

Neben den allgemeinen Methoden (siehe Tabelle 5.5), die alle Container-Klassen enthalten, bietet die Container-Klasse `list` noch viele zusätzliche Methoden an (siehe Tabellen 5.15 bis 5.20), die alle mit dieser Container-Klasse verwendet werden können. Die folgenden Tabellen sind wieder nach Themen sortiert. `T` steht erneut für den Typ.

Methode	Bedeutung
`T& front();`	Liefert eine Referenz auf das erste Element in der Liste zurück.
`T& back();`	Liefert eine Referenz auf das letzte Element in der Liste zurück.

Tabelle 5.15 Methoden für den Elementzugriff

Methode	Bedeutung
`void resize( size_type n, t = T() );`	Ändert die Container-Größe. Hierbei werden `n-size()` `t`-Elemente am Ende hinzugefügt bzw. `size()-n` Elemente gelöscht (abhängig davon, ob `n` größer oder kleiner als die aktuelle Größe ist).

Tabelle 5.16 Größenbezogene Methoden

Methode	Bedeutung
`void push_front( const T& obj );`	Fügt `obj` am Anfang der Liste ein.
`void push_back( const T& obj );`	Fügt `obj` hinter dem letzten Element in der Liste ein.
`iterator insert( iterator pos, const T& obj );`	Fügt das Objekt `obj` an der Position `pos` der Liste ein.
`void insert( iterator pos, size_type n, const T&obj );`	Fügt `n` Kopien vom Objekt `obj` an der Position `pos` der Liste ein.
`template <class InputIterator> void insert( iterator pos, InputIterator first, InputIterator last );`	Fügt die Elemente im Bereich `[first, last)` vor der Position `pos` ein. `first` und `last` zeigen nicht auf den Container, für den `insert` aufgerufen wird.

Tabelle 5.17 Methoden zum Einfügen neuer Elemente

Methode	Bedeutung
`void pop_front();`	Erstes Element in der Liste löschen.
`void pop_back();`	Letztes Element in der Liste löschen.
`iterator erase(iterator pos);`	Löscht das Element mit der Position `pos`. Der zurückgegebene Iterator verweist auf den Nachfolger von `pos` oder auf `end()`, falls das letzte Element gelöscht wurde.
`iterator erase( iterator first, iterator last );`	Löscht die Elemente im Bereich `[first,last)`. Der zurückgegebene Iterator verweist auf den Nachfolger von `last` oder auf `end()`, falls `last` nicht existiert.
`void clear();`	Löscht alle Listenelemente, so dass die Liste die Länge 0 besitzt.

Tabelle 5.18 Methoden zum Löschen von Elementen

Methode	Bedeutung
`list<T, Allocator>& operator=( const list<T, Allocator>& lis );`	Zuweisung an die aktuelle Liste (`*this`) mit dem kompletten Inhalt von `lis` (beispielsweise `mylis = lis`)

Tabelle 5.19 Zuweisen und Tauschen von Listen

Methode	Bedeutung
`template <class InputIterator> void assign(InputIterator first, InputIterator last);`	Container löschen und anschließend die Elemente aus dem Iterator-Bereich `[first,last)` einfügen.
`void assign( size_type n, const T& v );`	Container löschen und gegen `n` Elemente von `v` ersetzen.
`void swap( list<T, Allocator>& vect );`	Inhalte von zwei Listen vertauschen.
`allocator_type get_allocator() const;`	Liefert ein Objekt vom Typ der aktuell verwendeten Allokator-Klasse zurück.

Tabelle 5.19 Zuweisen und Tauschen von Listen (Forts.)

Methode	Bedeutung
`void splice( iterator pos, list<T,Allocator>& l );`	Fügt den Inhalt der Liste `l` vor `pos` ein. Die Liste `l` ist anschließend leer.
`void splice( iterator pos, list<T,Allocator>& l, iterator i );`	Fügt das Element aus der Position `i` von `l` vor der Position `pos` ein und entfernt das Element `i` aus `l`.
`void splice( iterator pos, list<T,Allocator>& l, iterator first, iterator last );`	Fügt die Elemente aus dem Bereich `[first,last)` aus `l` vor der Position `pos` ein und entfernt diese Elemente aus `l`. Falls die aktuelle Liste (`*this`) und `l` gleich sind, darf `pos` nicht im Bereich `[first,last)` sein.
`void merge( list<T, Allocator>& l );`	Mischen (Verschmelzen) zweier Listen. Sind die Listen sortiert, bleibt die zurückgegebene Liste (`*this`) ebenfalls sortiert. Die Liste `l` ist anschließend leer.
`template <class Compare> void merge( list<T, Allocator>& l, Compare comp );`	Wie zuvor, nur wird für den Vergleich von Elementen ein Compare-Objekt verwendet.
`void remove( const T& val );`	Entfernt alle Elemente mit dem Wert `val` aus der Liste.
`template <class Predicate> void remove_if(Predicate pred );`	Entfernt alle Elemente, auf die das Prädikat `pred` zutrifft (`true`). `Predicate` ist ein Funktionsobjekt mit einem Parameter für das Objekt `T`, das den Wert `true` oder `false` zurückliefert.
`void unique();`	Löscht alle gleich aufeinanderfolgenden Elemente bis auf das erste in der Liste. In einer sortierten Liste gibt es anschließend keine gleichen aufeinanderfolgenden Elemente mehr.

Tabelle 5.20 Listentypische Operationen

Methode	Bedeutung

`template <class BinaryPredicate>`
`void unique(`
`  BinaryPredicate binpred );`
	Entfernt gleich aufeinanderfolgende Elemente, auf die das Prädikat `binpred` zutrifft (`true`). `BinaryPredicate` ist ein Funktionsobjekt mit zwei Parametern für das Objekt `T`, das den Wert `true` oder `false` zurückliefert.

`void sort();`	Sortiert die Liste. Als Sortierkriterium muss der Operator `<` für den Typ `T` defniert sein.
`template <class Compare>` `void sort( Compare comp );`	Wie zuvor, nur wird für den Vergleich von Elementen ein Compare-Objekt verwendet.
`void reverse();`	Kehrt die Reihenfolge der Elemente in der Liste um.

Tabelle 5.20 Listentypische Operationen (Forts.)

Die Verwendung der Methoden gestaltet sich recht einfach, man sollte daher ein wenig experimentieren, um sich damit vertraut zu machen. Auch hier wollen wir wieder ein Beispiel mit einigen Methoden verwenden. Da Sie sich ja bereits dreimal mit den Listen herumgeschlagen haben – in den Abschnitten 2.8.1, »Strukturen«, (prozedural), 4.8.9, »Fallbeispiel: Verkettete Listen«, (objektorientiert) und 5.2, »Klassen-Templates« –, soll jetzt das gleiche Beispiel nochmals aufgegriffen werden, nur diesmal mit STL.

Es soll also noch einmal das Beispiel verwendet werden, in dem Sie ein Klassen-Template `Knoten` erzeugt haben, das beliebige Elemente aufnehmen konnte. In diesem Beispiel (siehe Abschnitt 5.2) haben Sie einmal Objekte der Klasse `Daten` und einmal Objekte der Klasse `Temperatur` verwaltet. Dies wollen wir hier nochmals mit STL realisieren, nur dass Sie hierbei ein Klassen-Template `Knoten` nicht mehr benötigen und das Listing erheblich kürzer ist.

Hierzu nochmals die Klassen `Daten` und `Temperatur`, die hier um die Operatoren `operator<` (für das Sortieren mit der Methode `sort()`) und `operator==` (für das Löschen mit der Methode `unique()`) erweitert werden mussten. Zuerst die Klasse `Daten`:

```
// daten.h
#include <iostream>
using namespace std;
#ifndef _DATEN_H_
#define _DATEN_H_

class Daten {
private:
   int iwert;
public:
```

```
   // Konstruktor
   Daten( int iVal ): iwert(iVal) {
   // Zu Debug- bzw. Verständniszwecken ggf. entfernen
   cout << "Objekt [Daten] erzeugt\n";
   }
   // Destruktor
   ~Daten( ) {}
   int vergleichen( const Daten& );
   void anzeigen() const;
   int get_iwert() const { return iwert; }
   // Überladener <-Operator wird für sort() benötigt
   friend bool operator <( const Daten& Dobj1,
                           const Daten& Dobj2 ) {
      if(Dobj1.iwert < Dobj2.iwert)
         return true;
      else
         return false;
   }
   // Überladener ==-Operator für unique()
   friend bool operator == ( const Daten& Dobj1,
                             const Daten& Dobj2) {
      if(Dobj1.iwert == Dobj2.iwert)
         return true;
      else
         return false;
   }
};

// Zum Vergleichen der Eigenschaften zweier Objekte
int Daten::vergleichen( const Daten& d ) {
   if( iwert > d.iwert ) { return 1; };
   if( iwert < d.iwert ) { return -1;  };
   return 0;
}

void Daten::anzeigen() const {
   cout << iwert << "\n";
}
#endif
```

Jetzt noch die Klasse `Temperatur`:

```
// temperatur.h
#include <iostream>
#include <cstring>
```

```
using namespace std;
#ifndef _TEMPERATUR_H_
#define _TEMPERATUR_H_

class Temperatur {
private:
   int grad;
   char monat[20];
public:
   // Konstruktor
   Temperatur( int iVal, char* m="" ): grad(iVal){
   // Zu Debug- bzw. Verständniszwecken ggf. entfernen
   strncpy( monat, m, 20 );
   cout << "Objekt [Temperatur] erzeugt\n";
   }
   // Destruktor
   ~Temperatur( ) {}
   int vergleichen( const Temperatur& );
   void anzeigen() const;
   // Überladener <-Operator wird für sort() benötigt
   friend bool operator < ( const Temperatur& Tobj1,
                            const Temperatur& Tobj2 ) {
      if(Tobj1.grad < Tobj2.grad)
         return true;
      else
         return false;
   }
   // Überladener ==-Operator wird für unique benötigt
   friend bool operator == ( const Temperatur& Tobj1,
                             const Temperatur& Tobj2 ) {
      return (! (strcmp(Tobj1.monat, Tobj2.monat)) );
   }
};

// Zum Vergleichen der Eigenschaften zweier Objekte
int Temperatur::vergleichen( const Temperatur& d ) {
   if( grad > d.grad ) { return 1; };
   if( grad < d.grad ) { return -1;  };
   return 0;
}

void Temperatur::anzeigen() const {
   cout << monat << " : " << grad << "\n";
}
#endif
```

Jetzt das Hauptprogramm, das die Container-Klasse `list` anhand dieser beiden Klassen demonstriert. Natürlich wurden noch weitere Methoden im Programm zur Demonstration verwendet:

```
// stl_list.cpp
#include <list>
#include <iostream>
#include "daten.h"
#include "temperatur.h"
using namespace std;

void input_func( list<Temperatur>& tempList );
void input_func( list<Daten>& datenList );
//template <class T>
void alles_anzeigen( list<Daten>& ref1,
                     list<Temperatur>& ref2 );

int main() {
   // eine verkettete Liste für die Klasse Daten
   list<Daten> dElemente;
   // eine verkettete Liste für die Klasse Temperatur
   list<Temperatur> tElemente;

   input_func(dElemente);   // <Daten>
   input_func(tElemente);   // <Temperatur>

  // ***************************************
  // Ab hier werden zusätzlich Methoden demonstriert, was
  // so nicht in der ursprünglichen Version des Programms
  // gemacht wurde
  // ***************************************

   cout << "Listen werden sortiert ...\n";
   // Benötigt den Operator < für Daten
   dElemente.sort();
   // ... the same ...
   tElemente.sort();
   alles_anzeigen(dElemente, tElemente);

   // Komplette Liste auf einmal zuweisen
   list<Daten> dElementeNeu = dElemente;
   if( dElementeNeu.empty() )
      cout << "\ndElementeNeu ist leer\n";
   else
      cout << "\ndElementeNeu ist nicht leer\n";
```

```
   cout << "\nDie Reihenfolge wird umgedreht ...\n";
   // Reihenfolge umdrehen
   dElemente.reverse();
   tElemente.reverse();
   alles_anzeigen(dElemente, tElemente);

   cout << "\nDoppelte Einträge werden entfernt ...\n";
   // Doppelte Elemente als direkte Nachfolger
   // aus der Liste entfernen
   dElemente.unique();
   tElemente.unique();
   alles_anzeigen(dElemente, tElemente);

   cout << "\nListe <Daten> mischen ...\n";
   // Mischen zweier Listen mit anschließendem Sortieren
   dElemente.merge(dElementeNeu);
   dElemente.sort();
   alles_anzeigen(dElemente, tElemente);

   if( dElementeNeu.empty() )
      cout << "\ndElementeNeu ist leer\n";
   else
      cout << "\ndElementeNeu ist nicht leer\n";

   // Liste mit drei Elementen der Klasse Daten und dem
   // Wert 111 erzeugen
   list<Daten> dElementeGanzNeu(3, Daten(111));
   // Inhalt von dElementeGanzNeu am Anfang
   // von dElemente einfügen
   cout << "3 Elemente am Anfang von <Daten> einfügen ...\n";
   dElemente.splice(dElemente.begin(), dElementeGanzNeu);
   alles_anzeigen(dElemente, tElemente);
   return 0;
}

// Funktion zum Einlesen der Eigenschaften für Temperatur
void input_func( list<Temperatur>& tempList ) {
   int grad;
   char monat[20];
   for(;;) {
      cout << "Monat eingeben : ";
      if( (!(cin >> monat)) )
         strcpy( monat, "keine Angabe");
      cout << "Temperatur eingeben (99=Ende) : ";
      if( (!(cin >> grad)) || grad == 99 )
```

```
         break; // Ende
      tempList.push_back(Temperatur(grad, monat));
   }
}

// Funktion zum Einlesen der Eigenschaften für Daten
void input_func( list<Daten>& datenList ) {
   int iwert;
   for(;;) {
      cout << "Wert eingeben (0=Ende) : ";
      if( (!(cin >> iwert)) || iwert == 0 )
         break; // Ende
      datenList.push_back(Daten(iwert));
   }
}

void alles_anzeigen( list<Daten>& ref1,
                     list<Temperatur>& ref2 ){
   for( list<Daten>::iterator iter = ref1.begin();
        iter != ref1.end(); ++iter ) {
      iter->anzeigen();
   }
   for( list<Temperatur>::iterator iter = ref2.begin();
        iter != ref2.end(); ++iter ) {
      iter->anzeigen();
   }
}
```

Das Programm bei der Ausführung:

```
Wert eingeben (0=Ende) : 10
Wert eingeben (0=Ende) : 12
Wert eingeben (0=Ende) : 8
Wert eingeben (0=Ende) : 13
Wert eingeben (0=Ende) : 43
Wert eingeben (0=Ende) : 10
Wert eingeben (0=Ende) : 10
Wert eingeben (0=Ende) : 11
Wert eingeben (0=Ende) : 0

Monat eingeben : Januar
Temperatur eingeben (99=Ende) : 9
Monat eingeben : Februar
Temperatur eingeben (99=Ende) : 12
Monat eingeben : Februar
Temperatur eingeben (99=Ende) : 11
```

```
Monat eingeben : März
Temperatur eingeben (99=Ende) : 19
Monat eingeben : April
Temperatur eingeben (99=Ende) : 15
Monat eingeben : 0
Temperatur eingeben (99=Ende) : 99

Listen werden sortiert ...
8
10
10
10
11
12
13
43

Januar  : 9
Februar : 11
Februar : 12
April   : 15
März    : 19

dElementeNeu ist nicht leer

Die Reihenfolge wird umgedreht ...
43
13
12
11
10
10
10
8

März    : 19
April   : 15
Februar : 12
Februar : 11
Januar  : 9

Doppelte Einträge werden entfernt ...
43
13
12
```

```
11
10
8

März    : 19
April   : 15
Februar : 12
Januar  : 9

Liste <Daten> mischen ...
8
8
10
10
10
10
11
11
12
12
13
13
43
43
...

dElementeNeu ist leer
3 Elemente (111) am Anfang von <Daten> einfügen ...
111
111
111
8
8
10
10
10
10
11
11
12
12
13
13
```

```
43
43
...
```

Die Container-Klasse »deque«

Die Container-Klasse `deque` (engl.: *double-ended queue*) ist eine Klasse für zweiendige Schlangen. `deque` ist intern wie `vector` aufgebaut und wird ebenfalls durch ein Array dargestellt. Der Vorteil von `deque` liegt im Zugriff auf Elemente, die sich am Anfang oder Ende befinden, da hierfür nur O(1) an Zeit benötigt wird. Wird ein Element allerdings irgendwo in der Mitte eingefügt, so ist die Zeit immer O(n).

`deque` wird mit der Header-Datei <*deque*> eingebunden. Neben den Datentypen aus der Tabelle 5.4 bietet ein `deque` noch folgende weitere öffentliche Datentypen:

Datentyp	Bedeutung
`pointer`	Zeiger auf deque-Element
`const_pointer`	Zeiger auf deque-Element (nur lesend anwendbar)

Tabelle 5.21 Öffentliche Datentypen für »deque«

Die Deklaration der Container-Klasse hat folgende Syntax:

```
template<class T> class deque;
```

Neben den allgemeinen Methoden (siehe Tabelle 5.5), die alle Container-Klassen enthalten, bietet die Container-Klasse `deque` noch zusätzliche an, die alle mit dieser Container-Klasse verwendet werden können. Die folgenden Tabellen (5.22 bis 5.26) sind wieder sortiert nach Themen. `T` steht erneut für den Typ.

Methode	Bedeutung
`T& operator[](size_type n);`	Gibt eine Referenz auf das `n`. Element des Deques zurück (beispielsweise `T[n]`; ist `n>=size()`, dann ist das Verhalten undefiniert).
`T& at(size_type n);`	Wie `operator[]`; nur wenn `n>=size()`, dann wird eine Exception vom Typ `out_of_range` ausgeworfen.
`T& front();`	Liefert eine Referenz auf das erste Element im Deque zurück.
`T& back();`	Liefert eine Referenz auf das letzte Element im Deque zurück.

Tabelle 5.22 Methoden für den Elementzugriff

[»] **Hinweis**

Alle Methoden für den Elementzugriff bieten auch eine `const`-Version an, mit der nur noch lesend auf die Elemente zugegriffen werden darf.

Methode	Bedeutung
`void resize( size_type n, t = T() );`	Ändert die Containergröße. Hierbei werden `n-size()` `t`-Elemente am Ende hinzugefügt bzw. `size()-n` Elemente gelöscht (abhängig davon, ob `n` größer oder kleiner als die aktuelle Größe ist).

Tabelle 5.23 Größenbezogene Methoden

Methode	Bedeutung
`void push_front( const T& obj );`	Fügt `obj` am Anfang des Deques ein.
`void push_back( const T& obj );`	Fügt `obj` hinter dem letzten Element im Deque ein.
`iterator insert( iterator pos, const T& obj );`	Fügt das Objekt `obj` an der Position `pos` des Deques ein.
`void insert( iterator pos, size_type n, const T&obj );`	Fügt `n` Kopien vom Objekt `obj` an der Position `pos` des Deques ein.
`template <class InputIterator> void insert( iterator pos, InputIterator first, InputIterator last );`	Fügt die Elemente im Bereich `[first, last)` vor der Position `pos` ein. `first` und `last` zeigen nicht auf den Container, für den `insert` aufgerufen wird.

Tabelle 5.24 Methoden zum Einfügen neuer Elemente

Methode	Bedeutung
`void pop_front();`	Erstes Element im Deque löschen.
`void pop_back();`	Letztes Element im Deque löschen.
`iterator erase(iterator pos);`	Löscht das Element mit der Position `pos`. Der zurückgegebene Iterator verweist auf den Nachfolger von `pos` oder auf `end()`, falls das letzte Element gelöscht wurde.
`iterator erase( iterator first, iterator last );`	Löscht die Elemente im Bereich `[first,last)`. Der zurückgegebene Iterator verweist auf den Nachfolger von `last` oder auf `end()`, falls `last` nicht existiert.
`void clear();`	Löscht alle Deque-Elemente, so dass der Vektor die Länge 0 besitzt.

Tabelle 5.25 Methoden zum Löschen von Elementen

Methode	Bedeutung
`deque<T, Allocator>& operator=( const deque<T, Allocator>& deq );`	Zuweisung an aktuellen Deque (`*this`) mit dem kompletten Inhalt von `deq` (beispielsweise `mydeq = deq`)
`template <class InputIterator> void assign(InputIterator first, InputIterator last);`	Container löschen und anschließend die Elemente aus dem Iterator-Bereich `[first,last)` einfügen.
`void assign( size_type n, const T& v );`	Container löschen und gegen `n` Elemente von `v` ersetzen.
`void swap( deque<T, Allocator>& vect );`	Inhalte von zwei Deques vertauschen.

Tabelle 5.26 Zuweisen und Tauschen von Deques

Da die Klasse `deque` genauso wie die Klasse `vector` anzuwenden ist, wurde hier auf ein Beispiel verzichtet. Zum Testen können Sie ja das Listing *stl_vector.cpp* heranziehen und `vector` gegen `deque` austauschen (und natürlich auch die Iteratoren).

Abstrakte Container (Adapter-Klassen)

Unter einem Adapter versteht man die Implementierung eines abstrakten Datentyps (ADT), der als implizite Datenstruktur einen STL-Container verwendet. Der Adapter kapselt dabei die Funktionalität des verwendeten Containers und lässt nur den Zugriff auf die vom ADT benötigten Funktionen zu. Diese Funktionen heißen dann im Allgemeinen auch anders als die entsprechenden Container-Versionen.

Die Templates `stack` und `queue` sind nicht als eigenständige Container in STL definiert, sondern wurden aus anderen Containern modifiziert. Standardmäßig sind die Klassen `stack` und `queue` eine Modifikation von `deque`.

```
template <class T, class Container=deque<T> >
class stack {
   protected: Container c;
   ...
};

template <class T, class Container=deque<T> >
class queue {
   protected: Container c;
   ...
};
```

An der Syntax der beiden Container `stack` und `queue` lässt sich erkennen, dass Sie für die voreingestellte Container-Klasse `deque` auch einen anderen Container angeben können. Möglich wären hierfür `list` und `vector`. Man sagt auch, der Container wird *gelayert*.

Die `priority_queue` ist wie die `queue` eine Warteschlange, nur hat diese Klasse ein zusätzliches Attribut `Priorität`.

Solche Container-Klassen werden auch als *Adapter-Klassen* bezeichnet. Es wird praktisch eine vorhandene Schnittstelle (Standard: `deque`) verwendet, und ein Adapter schaltet zwischen dem Benutzer und den impliziten Datentypen hin und her. So verwendet man bei einem Stack-Objekt über die Stack-Methoden die darüberliegenden Container (Standard:`deque`).

Die Container-Klasse »stack«

Ein `stack` ist ein Container, der zur Ablage und Entnahme von einer Seite verwendet wird. Wird praktisch ein Element oben (oder korrekterweise unten, wie man will) auf einem Stack abgelegt (`push`), ist dieses Element das erste, das anschließend wieder entnommen (`pop`) werden kann. Dieses Prinzip wird LIFO-Prinzip genannt (*LIFO = Last In First Out*). Zur Verwendung der Container-Klasse `stack` wird die Header-Datei <*stack*> benötigt.

Wie bereits erwähnt, wird der `stack` intern mit einem anderen Container realisiert (standardmäßig: `deque`). Hierfür können auch die Container-Klassen `list` und `vector` verwendet werden. Das kann dann zum Beispiel so realisiert werden:

```
// Standardmäßig mit deque realisierter Stack
stack<int> Stack1;
// mit vector realisierter Stack
stack<int, vector<int> > Stack2;
// mit list realisierter Stack
stack<int, list<int> > Stack3;
```

Wenn Sie eine andere Container-Klasse als die standardmäßig eingestellte verwenden, beachten Sie bitte, dass Sie ein Leerzeichen am Ende der spitzen Klammern einfügen, weil der Compiler sonst beanstanden kann, dass es sich hier um den Operator >> handelt.

Die Deklaration der Klasse sieht wie folgt aus:

```
template <class T, class Container = deque<T> > class stack;
```

Ein Stack stellt Ihnen die in Tabelle 5.27 gezeigten Datentypen zur Verfügung:

Datentyp	Bedeutung
`value_type`	Typ `T` der Elemente
`size_type`	integraler Typ ohne Vorzeichen für Größenangaben
`container_type`	Typ des Containers (`deque`, `vector` oder `list`)

Tabelle 5.27 Datentypen für die Container-Klasse »stack«

Da bei einem Stack nicht viele Aktionen nötig sind, sind auch nicht viele Methoden vorhanden (siehe Tabelle 5.28). Außerdem sind die Stack-Methoden `top()`, `push()` und `pop()` direkt auf die Operationen `push_back()`, `pop_back()` und `back()` des zugrunde liegenden Containers (`deque`, `vector` und `list` haben solche Methoden) zurückzuführen. `T` steht wieder für den Typ.

Methode	Bedeutung
`explicit stack(const Container&= Container() );`	Konstruktor. Ein Stack kann mit einem bereits vorhandenen Container initialisiert werden bzw. ein Stack mit der Länge 0. `Container` ist der Typ des Containers.
`T& top();`	Gibt das oberste (bzw. unterste) Element des Stacks zurück. Diese Methode gibt es auch als `const`-Version.
`void push( const T& s );`	Fügt das Objekt `s` hinter dem letzten Element des Stacks ein.
`void pop();`	Löscht das zuletzt eingefügte Element im Stack (also das Element, das oben auf dem Stapel liegt).

Tabelle 5.28 Mögliche Methoden für die Container-Klasse »stack«

Natürlich sind hier auch die relationalen Operatoren `!=`, `<=`, `>=` usw. vorhanden.

Hierzu ein einfaches Beispiel, das die Adapter-Klasse `stack` in der Praxis demonstrieren soll:

```
// stl_stack.cpp
#include <stack>
#include <list>
#include <vector>
#include <iostream>
using namespace std;

class A {
private:
   int ival;
   float fval;
```

```
public:
   // Konstruktor
   A( int i=0, float f=0.0f ): ival(i), fval(f) {}
   // Destruktor
   ~A( ) {}
   void anzeigen() const {
      cout << ival << " : " << fval << "\n";
   }
};

int main() {
   // Ein Stack mit int-Werten
   // Standardmäßig mit deque realisiert
   stack<int> Stack1;
   // Ein Stack mit Objekten der Klasse A
   // Stack mit Container-Klasse list realisieren
   stack<A, list<A> > Stack2;
   // Ein Stack mit Zeichenketten
   // Stack mit der Container-Klasse vector realisiert
   stack<char*, vector<char*> > Stack3;

   // Elemente mit push auf den Stack einfügen
   for( int i = 1; i < 10; i++ )
      Stack1.push(i);
   float f=1.1f;
   for( int i=1; i < 10; i++ ) {
      Stack2.push( A(i,f) );
      f+=1.1f;
   }
   Stack3.push("Tag");
   Stack3.push("schöner ");
   Stack3.push("ist ");
   Stack3.push("Heute ");

   // Elemente vom Stack1 ausgeben und wieder entfernen
   while( Stack1.empty() != true ) {
      cout << Stack1.top() << " ";
      // Oberstes Element entfernen
      Stack1.pop();
   }
   cout << endl;
   // Elemente vom Stack2 ausgeben und wieder entfernen
   while( Stack2.empty() != true ) {
      A& tmp = Stack2.top();
      tmp.anzeigen();
      // Oberstes Element entfernen
      Stack2.pop();
```

```
   }
   cout << endl;
   // Elemente vom Stack1 ausgeben und wieder entfernen
   while( Stack3.empty() != true ) {
      cout << Stack3.top() << " ";
      // Oberstes Element entfernen
      Stack3.pop();
   }
   cout << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
9 8 7 6 5 4 3 2 1
9 : 9.9
8 : 8.8
7 : 7.7
6 : 6.6
5 : 5.5
4 : 4.4
3 : 3.3
2 : 2.2
1 : 1.1

Heute  ist  schöner  Tag
```

Die Container-Klasse »queue«

Die Adapter-Klasse `queue` (oder auch Warteschlange) wird verwendet, um ein Objekt auf der einen Seite abzulegen (`push`) und um ein anderes Objekt von der anderen Seite zu entnehmen (`pop`). Dieses Prinzip wird FIFO-Prinzip genannt (*FIFO = First In First Out*). Auch die `queue` ist mit der Container-Klasse `deque` realisiert, aber hier würde sich auch die Container-Klasse `list` sehr gut eignen. Natürlich kann dies auch mit der Container-Klasse `vector` gemacht werden, aber da beim Einfügen hinten alle Elemente nach vorn verschoben werden müssen, ist sie weniger effizient.

Wenn Sie eine `queue` verwenden wollen, müssen Sie die Header-Datei <*queue*> mit einbinden. Sofern Sie die Adapter-Klasse `queue` nicht mit `deque`, sondern mit `list` realisieren wollen, gehen Sie, wie schon beim `stack`, folgendermaßen vor:

```
// Standardmäßig mit deque realisierte Queue
queue<int> Queue1;
// mit list realisierte Queue
queue<int, list<int> > Queue2;
```

Die Deklaration der Klasse sieht folgendermaßen aus:

```
template<class T, class Container = deque<T> > class queue;
```

Eine Queue stellt die in der Tabelle 5.29 gezeigten Datentypen zur Verfügung:

Datentyp	Bedeutung
`value_type`	Typ `T` der Elemente
`size_type`	integraler Typ ohne Vorzeichen für Größenangaben
`container_type`	Typ des Containers (`deque`, `vector` oder `list`)

Tabelle 5.29 Datentypen für die Container-Klasse »queue«

Auch bei Queue werden die Methoden `pop()` und `push()` einfach mit den Methoden `push_back()` und `pop_front()` des zugrunde liegenden Containers verwendet. Hierzu die einzelnen Methoden, die für die Verwendung einer `queue` benötigt werden:

Methode	Bedeutung
`explicit queue( const Container&=Container());`	Konstruktor. Eine Queue kann mit einem bereits vorhandenen Container initialisiert werden bzw. eine Queue mit der Länge 0. `Container` ist der Typ des Containers.
`T& back();`	Gibt das letzte Element der Queue zurück. Diese Methode gibt es auch als `const`-Version.
`T& front();`	Gibt das erste Element der Queue zurück. Diese Methode gibt es auch als `const`-Version.
`void push( const T& s );`	Fügt das Objekt `s` hinter dem letzten Element der Queue ein.
`void pop();`	Löscht das erste Element in der Queue.

Tabelle 5.30 Mögliche Methoden für die Container-Klasse »queue«

Des Weiteren sind hier auch die relationalen Operatoren `!=`, `<=`, `>=` usw. vorhanden.

Es folgt ein einfaches Beispiel, das die Adapter-Klasse `queue` in der Praxis demonstrieren soll. Das Beispiel ist dasselbe, das Sie schon bei *stl_stack.cpp* kennengelernt haben, nur mit einigen Modifikationen für `queue`. Die Ausgabe des Programms ist genau umgekehrt wie beim Stack:

```
// stl_queue.cpp
#include <queue>
#include <list>
#include <iostream>
```

```
using namespace std;

class A {
private:
   int ival;
   float fval;
public:
   // Konstruktor
   A( int i=0, float f=0.0f ): ival(i), fval(f) {}
   // Destruktor
   ~A( ) {}
   void anzeigen() const {
      cout << ival << " : " << fval << "\n";
   }
};

int main() {
   // Ein Queue mit int-Werten
   // Standardmäßig mit deque realisiert
   queue<int> Queue1;
   // Ein Queue mit Objekten der Klasse A
   // Queue mit Container-Klasse list realisieren
   queue<A, list<A> > Queue2;
   // Eine Queue mit char-Werten
   queue<char*> Queue3;

   // Elemente mit push am Ende der Queue einfügen
   for( int i = 1; i < 10; i++ )
      Queue1.push(i);
   float f=1.1f;
   for( int i=1; i < 10; i++ ) {
      Queue2.push( A(i,f) );
      f+=1.1f;
   }
   Queue3.push("Tag ");
   Queue3.push("schöner ");
   Queue3.push("ist ");
   Queue3.push("Heute ");

   // Elemente von Queue1 ausgeben und wieder entfernen
   while( Queue1.empty() != true ) {
      cout << Queue1.front() << " ";
      // Letztes Element entfernen
      Queue1.pop();
   }
```

```
   cout << endl;
   // Elemente von Queue2 ausgeben und wieder entfernen
   while( Queue2.empty() != true ) {
      A& tmp = Queue2.front();
      tmp.anzeigen();
      // Oberstes Element entfernen
      Queue2.pop();
   }
   cout << endl;
   // Elemente vom Stack1 ausgeben und wieder entfernen
   while( Queue3.empty() != true ) {
      cout << Queue3.front() << " ";
      // Oberstes Element entfernen
      Queue3.pop();
   }
   cout << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
1 2 3 4 5 6 7 8 9
1 : 1.1
2 : 2.2
3 : 3.3
4 : 4.4
5 : 5.5
6 : 6.6
7 : 7.7
8 : 8.8
9 : 9.9

Tag  schöner  ist  Heute
```

Die Container-Klasse »priority_queue«

Eine Priority-Queue ist wie die Queue einer Warteschlange, nur dass diese noch zusätzlich ein Attribut `Priorität` besitzt. Jedem Element wird hierbei schon beim Einfügen eine Priorität zugeordnet. Die relative Priorität wird dann durch den Vergleich zweier Elemente bestimmt. Hierzu wird entweder der Operator < oder ein Funktionsobjekt – siehe Abschnitt 5.3.2, »Hilfsmittel (Hilfsstrukturen)« – verwendet. Wenn Sie die Priority-Queue mit Klassen (was gewöhnlich der Fall ist) verwenden wollen, müssen Sie den Operator < in der Klasse implementieren. Das Element mit der höchsten Priorität ist immer das erste Element, das von der `priority_queue` entnommen wird – also ähnlich wie bei einem Stack.

Natürlich wurde auch die `priority_queue` mit `deque` realisiert. Aber auch die Container-Klasse `vector` ist für die Implementierung geeignet. Die Deklaration der Klasse sieht wie folgt aus:

```
template<class T, class Container = deque<T>,
         class Compare =
            less<typename Container::value_type> >
class priority_queue;
```

Wenn Sie eine `priority_queue` verwenden wollen, müssen Sie die Header-Datei <*queue*> mit einbinden. Sofern Sie die Adapter-Klasse `priority_queue` nicht mit `deque`, sondern mit `vector` realisieren wollen, gehen Sie folgendermaßen vor:

```
// Priority-Queue wird mit einem Vector statt einer
// Deque realisiert
priority_queue<int, vector<int> > Pqueue1;
```

Bei der Deklaration der Klasse `priority_queue` kann man auch erkennen, dass die Priorität mit dem Funktionsobjekt `less` realisiert wird – siehe Abschnitt 5.3.2, »Hilfsmittel (Hilfsstrukturen)«. Das bedeutet in der Praxis, dass die Elemente mit dem »höheren« Wert höchste Priorität haben. Wollen Sie das Gegenteil bewirken, müssen Sie das Funktionsobjekt `greater` verwenden:

```
// Funktionsobjekt greater ==
// kleines Element -> hohe Priorität
priority_queue<int, vector<int>, greater<int> > PQueue2;
```

Natürlich können Sie auch andere vordefinierte oder gar selbstgeschriebene Funktionsobjekte (siehe Abschnitt 5.3.2) verwenden. Die normale Deklaration einer `priority_queue` lässt sich wie gehabt realisieren:

```
priority_queue<int> Pqueue3;
priority_queue<A> Pqueue4;
```

Für die Priority-Queue sind übrigens keine globalen Operatoren definiert, weil hier ein Vergleich eigentlich keinen Sinn machen würde. Nun aber zu den Methoden, die im Zusammenhang mit einer Priority-Queue verwendet werden:

Methoden	Bedeutung
`explicit priority_queue( const Compare& =Compare(), const Container& =Container() );`	Konstruktor. Eine Priority-Queue kann mit einem bereits vorhandenen Container initialisiert werden bzw. eine Queue mit der Länge 0. `Container` ist der Typ des Containers.

Tabelle 5.31 Methoden für die Adapter-Klasse »priority_queue«

Methoden	Bedeutung
`template class<InputIterator> priority_queue( InputIterator first, InputIterator last, const Compare& =Compare(), const Container& =Container() );`	Der Konstruktor unterscheidet sich von dem vorhergehenden, indem die Elemente im Bereich `(first, last]` eines anderen Containers bei der Konstruktion zusätzlich eingefügt werden.
`void push( const T& s );`	Fügt das Objekt `s` in die Priority-Queue ein.
`void pop();`	Löscht das Element mit der höchsten Priorität in der Priority-Queue.
`const T& top() const;`	Gibt das Element mit der höchsten Priorität zurück.

Tabelle 5.31 Methoden für die Adapter-Klasse »priority_queue« (Forts.)

Hierzu ein einfaches Beispiel, das die Priority-Queue in der Praxis demonstrieren soll:

```
// stl_priority_queue.cpp
#include <iostream>
#include <queue>
using namespace std;

class A {
private:
   int ival;
   float fval;
public:
   // Konstruktor
   A( int i=0, float f=0.0f ): ival(i), fval(f) {}
   // Destruktor
   ~A( ) {}
   void anzeigen() const {
      cout << ival << " : " << fval << "\n";
   }
   // Überladener <-Operator wird für die
   // Priorität-Warteschlange benötigt
   friend bool operator < (const A& ival1, const A& ival2) {
      if(ival1.ival < ival2.ival)
         return true;
      else
         return false;
   }
};
```

```
int main() {
   // Standardmäßig mit der Priorität less ==
   // große Elemente -> hohe Priorität
   priority_queue<int> PQueue1;
   // Funktionsobjekt greater ==
   // kleines Element -> hohe Priorität
   priority_queue<int, vector<int>, greater<int> > PQueue2;
   priority_queue<A> PQueue3;
   int vec[] = { 12, 11, 3, 44, 55 };

   for( int i=0; i < sizeof(vec)/sizeof(int); i++ )
      PQueue1.push(vec[i]);
   for( int i=0; i < sizeof(vec)/sizeof(int); i++ )
      PQueue2.push(vec[i]);
   float f=0.0f;
   for( int i=0; i < sizeof(vec)/sizeof(int); i++ ) {
      PQueue3.push( A(vec[i], f) );
      f+=1.1;
   }

   while( PQueue1.empty() != true ) {
      cout << PQueue1.top() << " ";
      PQueue1.pop();
   }
   cout << endl;
   while( PQueue2.empty() != true ) {
      cout << PQueue2.top() << " ";
      PQueue2.pop();
   }
   cout << endl;
   while( PQueue3.empty() != true ) {
      const A& tmp = PQueue3.top();
      tmp.anzeigen();
      PQueue3.pop();
   }
   cout << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
55 44 12 11 3

3 11 12 44 55
```

```
55 : 4.4
44 : 3.3
12 : 0
11 : 1.1
3  : 2.2
```

Assoziative Container

Die Container, die Sie bisher verwendet haben, konnten die Elemente immer nur der Reihe nach ablegen. Bei einem assoziativen Container ist dies anders. Der Zugriff auf die Elemente eines Containers erfolgt hier mit einem frei wählbaren Schlüssel (Key). Damit erfolgt der Zugriff auf die einzelnen Elemente in einem assoziativen Container nicht mehr über den Index oder einen Iterator, sondern über diesen Schlüssel, der nicht notwendigerweise mit den Daten übereinstimmen muss. Die STL bietet mit `set`, `multiset`, `map` und `multimap` vier Arten von assoziativen Containern an, die auf den nächsten Seiten etwas näher erläutert werden.

Die Container-Klassen »set« und »multiset«

Die in einem `set`- bzw. `multiset`-Container eingefügten Elemente werden sofort in eine sortierte Reihenfolge gebracht. Beide Container definieren einen sortierten Container, der nur Schlüssel enthält. Der Unterschied zwischen `set` und `multiset` liegt darin, dass in `set` ein Schlüssel (Element) nur einmal vorkommen darf. Bei `multiset` darf es mehrere gleiche Schlüssel (Elemente) geben. Würden Sie versuchen, bei `set` einen Schlüssel (Element) einzufügen, der bereits vorhanden ist, so würde dieses Einfügen nicht durchgeführt werden, weil die Regel von `set` besagt, dass ein Schlüssel nur einmal vorkommen darf. Für den anderen Fall gibt es `multiset`.

Intern wird `set` bzw. `multiset` gewöhnlich über binäre Bäume realisiert (was allerdings nicht so sein muss).

Im mathematischen Sinn liegen zwar die einzelnen Schlüssel (Elemente) einer Menge (`set` = Menge) nicht in einer sortierten Ordnung vor, aber da dies den Zugriff auf den Schlüssel erheblich erleichtert, wird die Menge intern doch geordnet. Auch hier wird als Ordnungskriterium wieder standardmäßig das Funktionsobjekt `less<T>` (Sortierung aufsteigend) verwendet, was aber auch hier wieder explizit verändert werden kann.

`set` bzw. `multiset` ist die ideale Struktur, wenn Sie schnell ein Element in einem Container ablegen wollen und auch schnell wieder darauf zugreifen müssen. Im Vergleich verläuft das Einfügen eines neuen Elements zwar genauso effektiv ab wie bei einer Liste (`list`), aber beim Suchen nach einem entsprechenden Ele-

ment ist `set` bzw. `multiset` erheblich schneller (da binärer Baum). Auch das nachträgliche Sortieren einer Liste entfällt bei `set` bzw. `multiset` komplett.

Damit Sie mit der Container-Klasse `set` bzw. `multiset` arbeiten können, müssen Sie die Header-Datei *<set>* mit einbinden. Die Deklaration der Klasse sieht folgendermaßen aus:

```
Template <class Key,                    // Schlüssel
          class Compare = less<Key> > // für Sortierung
class set;
```

Für die Sortierung wird standardmäßig der <-Operator (also `less<Key>`) verwendet. Beim Einsatz von Elementen, die Klassen sind, müssen Sie dementsprechend den Operator < implementieren, oder Sie können auch eine Klasse für Funktionsobjekte angeben. Somit kann die Definition eines `set` folgendermaßen aussehen:

```
// ein leeres set für int
set <int> Set1;

// Ein set mit Klasse als Funktionsobjekt
class FuncObj { ....};  // Klasse für Funktionsobjekt
...
set<int, FuncObj> Set2;

// auch mit vordefinierten Funktionsobjekten möglich
set<int, greater<int> > Set3;

// ... oder last but not least: Eine komplette Liste in ein
// set umkopieren ...
list<int> aList;
...
set<int> Set4(aList.begin(), aList.end());
```

Hinweis [«]

Alle hier gezeigten Definitionen sind natürlich auch für `multiset` anwendbar.

Neben den bereits gezeigten öffentlichen Datentypen (siehe Tabelle 5.4) stehen Ihnen für die Klasse `set` bzw. `multiset` folgende weitere Datentypen zur Verfügung:

Datentyp	Bedeutung
`pointer`	Zeiger auf `set`-Element
`const_pointer`	Zeiger auf `set`-Element, der nur lesend verwendet werden kann

Tabelle 5.32 Weitere Datentypen für die Container »set« und »multiset«

Datentyp	Bedeutung
`key_type`	Key
`value_type`	wie Key
`key_compare`	Compare (Vorgabe `less<T>`)
`value_compare`	Compare (Vorgabe `less<T>`)

Tabelle 5.32 Weitere Datentypen für die Container »set« und »multiset« (Forts.)

Abgesehen von den für alle Container vorhandenen Methoden stehen dem Container `set` bzw. `multiset` noch folgende Methoden zur Verfügung (siehe Tabellen 5.33 bis 5.37):

Methoden	Bedeutung
`explicit set( const Compare& comp=Compare(), const Allocator&=Allocator() );`	Erzeugt ein `set` mit der Länge 0.
`template class<InputIterator> set( InputIterator first, InputIterator second, const Compare& comp=Compare(), const Allocator&=Allocator() );`	Erzeugt ein `set` mit der Länge 0 und fügt anschließend die Elemente `[first,last)` ein.
`set( const set< T, Compare, Allocator>& s );`	Erzeugt ein `set` als Kopie von `s`.
`~set();`	Zerstört ein `set` (gibt es frei).

Tabelle 5.33 Konstruktoren und Destruktor von »set« bzw. »multiset«

[»] **Hinweis**

Zwar wird hier vorwiegend die Syntax von `set` verwendet, die Syntax von `multiset` ist aber gleich, nur dass `multiset` statt `set` geschrieben wird und `multiset` mehrere gleiche Schlüssel erlaubt.

Methode	Bedeutung
`iterator erase(iterator pos);`	Löscht das Element mit der Position `pos`. Der zurückgegebene Iterator verweist auf den Nachfolger von `pos` oder auf `end()`, falls das letzte Element gelöscht wurde.
`iterator erase( iterator first, iterator last );`	Löscht die Elemente im Bereich `[first,last)`. Der zurückgegebene Iterator verweist auf den Nachfolger von `last` oder auf `end()`, falls `last` nicht existiert.

Tabelle 5.34 Methoden zum Löschen von Elementen

Methode	Bedeutung
`size_type erase( const T& s );`	Löscht das Objekt `s` (falls vorhanden). Rückgabewert `1` = gelöscht; `0` = nicht gelöscht. Bei `multiset` werden alle Elemente `s` gelöscht.
`void clear();`	Löscht alle `set`-Elemente, so dass `set` die Länge 0 besitzt.

Tabelle 5.34 Methoden zum Löschen von Elementen (Forts.)

Methode	Bedeutung
`pair<iterator, bool> insert( const T& obj );`	Das Objekt `obj` wird eingefügt, falls es noch nicht existiert (gilt nicht für `multiset`). Der Iterator des zurückgegebenen Paares zeigt auf den eingefügten Schlüssel bzw. auf den schon vorhandenen Schlüssel mit demselben Wert wie `obj`. Der boolesche Wert zeigt, ob überhaupt ein Einfügen stattgefunden hat (es kann ja sein, dass der Schlüssel bereits existiert).
`iterator insert( iterator pos, const T& obj );`	Fügt das Objekt `obj` im Set ein, falls `obj` noch nicht existiert (gilt nicht für `multiset`). Die Position `pos` ist lediglich ein Hinweis, ab wo eingefügt werden soll (die Objekte werden ja sowieso sortiert).
`template <class InputIterator> void insert( InputIterator first, InputIterator last );`	Fügt die Elemente des Bereichs `[first, last)` eines anderen Containers in den aktuellen ein. Auch hierbei werden nur die Elemente eingefügt, die noch nicht vorhanden sind (gilt nicht für `multiset`).

Tabelle 5.35 Methoden zum Einfügen neuer Elemente

Methoden	Bedeutung
`iterator find( const T& s ) const;`	Sucht nach dem Objekt `s` im `set`. Der Rückgabewert ist die Position von `s` oder `end()`, falls `s` nicht vorhanden ist.
`iterator lower_bound( const T& s ) const;`	Gibt die Position des ersten Objekts im Set zurück, das größer gleich `s` ist (auch als `const`-Version vorhanden).
`iterator upper_bound( const T& s ) const;`	Gibt die erste Position des Objekts im Set zurück, das nicht kleiner gleich `s` ist (auch als `const`-Version vorhanden).
`pair<iterator, iterator> equal_range( const T& s ) const;`	Liefert die beiden Werte `upper_bound` und `lower_bound` als Paar.

Tabelle 5.36 Suchoperationen für »set«

Methoden	Bedeutung
`size_type count ( const T& s ) const;`	Gibt `1` zurück, wenn `s` im Set vorkommt, ansonsten `0`. Bei `multiset` gibt diese Methode die Anzahl zurück, die `s` im `multiset` vorhanden ist.

Tabelle 5.36 Suchoperationen für »set« (Forts.)

Methoden	Bedeutung
`void swap( set<T, Compare, Allocator>& s );`	Den Inhalt des Sets `*this` mit `s` vertauschen.
`set<T, Compare, Allocator>& operator=( const set<T, Compare, Allocator>& s );`	Zuweisung eines Sets `s` an das Set `*this`.
`key_compare key_comp() const`	Gibt eine Kopie des Vergleichsobjekts zurück, das bei der Erzeugung der `map` verwendet wurde.
`value_compare value_comp() const`	Wie `key_comp()`. Hat erst bei `map` eine Bedeutung.

Tabelle 5.37 Weitere Methoden

[»] Hinweis

In STL gibt es eine allgemeine Funktion `find()`, die für alle Container geeignet ist, und auch `set` bietet eine eigene Funktion `find()`. Die Elementfunktion von `set` ist hierbei allerdings wesentlich schneller als die allgemeine.

Dazu ein Beispiel, das die Methoden von `set` und `multiset` in der Praxis demonstrieren soll:

```
// stl_set.cpp
#include <iostream>
#include <set>
using namespace std;

template <class Container>
void showSet(const Container& s );

class A {
private:
   int ival;
   float fval;
public:
   // Konstruktor
   A( int i=0, float f=0.0f ): ival(i), fval(f) {}
   // Destruktor
   ~A( ) {}
```

```
   void anzeigen() const {
      cout << ival << " : " << fval << "\n";
   }
   int get_ival() const { return ival; }
   float get_fval() const { return fval; };
   // Überladener <-Operator
   friend bool operator < (const A& ival1, const A& ival2) {
      if(ival1.ival < ival2.ival)
         return true;
      else
         return false;
   }
};

int main() {
   // für den Vergleich standardmäßig Funktionsobjekt less
   set<int> Set1;
   // für den Vergleich Funktionsobjekt greater
   set<int, greater<int> > Set2;
   set<A> Set3;
   // für den Vergleich standardmäßig less
   multiset<int> Mset1;
   multiset<A> Mset2;

   for(int i = 0; i < 10; ++i)
      Set1.insert(i);
   for(int i = 0; i < 10; ++i)
      Set2.insert(i);
   float f=0.0f;
   for(int i = 0; i < 10; ++i) {
      Set3.insert(A(i, f));
      f+=1.1f;
   }
   cout << "Set1  : ";
   showSet( Set1 );
   cout << "Set2  : ";
   showSet( Set2 );
   cout << "Set3  : \n ";
   for( set<A>::const_iterator iter = Set3.begin();
        iter != Set3.end(); iter++ )
      iter->anzeigen();

   // Versuch, dieselben Elemente nochmals in Set1
   // einzufügen, ist wirkungslos, weil die Elemente
   // bei set exakt nur einmal vorkommen ...
```

```
// ... die Werte 10 und 11 werden hier aber übernommen
for(int i = 0; i < 12; ++i)
   Set1.insert(i);
cout << "Set1  : ";
showSet( Set1 );

// ... im Gegensatz zu set kann es bei multiset mehrere
// gleiche Schlüssel (Elemente) geben ...
for(int i = 0; i < 10; ++i)
   Mset1.insert(i);
for(int i = 0; i < 12; ++i)
   Mset1.insert(i);
f=0.0f;
for(int i = 0; i < 10; ++i) {
   Mset2.insert(A(i,f));
   f+=1.1f;
}
f=0.0f;
for(int i = 0; i < 12; ++i) {
   Mset2.insert(A(i,f));
   f+=1.1f;
}
cout << "\nMset1 : ";
showSet(Mset1);
cout << "Schlüssel/Element 3 kommt " << Mset1.count(3)
     << "mal im Multiset Mset1 vor\n";
cout << "Mset2  : \n ";
for( set<A>::const_iterator iter = Mset2.begin();
     iter != Mset2.end(); iter++ )
   iter->anzeigen();
cout << "Schlüssel/Element A(3, 3.3) kommt "
     << Mset2.count(A(3, 3.3))
     << "mal im Multiset Mset2 vor\n";

int val;
cout << "Element suchen (0...11) : ";
cin >> val;
set<int>::const_iterator iter = Set1.find(val);
if( iter != Set1.end() )
   cout << "Element " << val << " gefunden\n";
else
   cout << "Element " << val << " nicht gefunden\n";

int ival;
float fval;
```

```
   cout << "Integer-Wert zum Suchen   : ";
   cin >> ival;
   cout << "Gleitpunktzahl zum Suchen : ";
   cin >> fval;
   set<A>::const_iterator iter2 = Mset2.find(A(ival, fval));
   if( iter2 != Mset2.end() )
      if( iter2->get_ival() == ival &&
          iter2->get_fval() == fval )
         cout << "100%ige Übereinstimmung -> gefunden\n";
      else
         cout << "Nur der erste Wert stimmt\n";
   else
      cout << "Element nicht gefunden\n";

   cout << "Welches Element löschen (0...11) : ";
   cin >> val;
   int cnt = Mset1.erase(val);
   if( cnt != 0 )
      cout << cnt << " Elemente gelöscht\n";
   else
      cout << val << " ist nicht vorhanden\n";
   cout << "Mset1 : ";
   showSet( Mset1 );
   return 0;
}

template<class Container>
void showSet(const Container& s ) {
   typename Container::const_iterator iter = s.begin();
   while(iter != s.end())
      cout << *iter++ << " ";
   cout << endl;
}
```

Das Programm bei der Ausführung:

```
Set1  : 0 1 2 3 4 5 6 7 8 9
Set2  : 9 8 7 6 5 4 3 2 1 0
Set3  :
0 : 0
1 : 1.1
2 : 2.2
3 : 3.3
4 : 4.4
5 : 5.5
```

```
6 : 6.6
7 : 7.7
8 : 8.8
9 : 9.9
Set1  : 0 1 2 3 4 5 6 7 8 9 10 11

Mset1 : 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 11
Schlüssel/Element 3 kommt 2mal im Multiset Mset1 vor
Mset2  :
0 : 0
0 : 0
1 : 1.1
1 : 1.1
2 : 2.2
2 : 2.2
3 : 3.3
3 : 3.3
4 : 4.4
4 : 4.4
5 : 5.5
5 : 5.5
6 : 6.6
6 : 6.6
7 : 7.7
7 : 7.7
8 : 8.8
8 : 8.8
9 : 9.9
9 : 9.9
10 : 11
11 : 12.1
Schlüssel/Element A(3, 3.3) kommt 2mal im Multiset Mset2 vor
Element suchen (0...11) : 5
Element 5 gefunden
Integer-Wert zum Suchen   : 2
Gleitpunktzahl zum Suchen : 2.2
100%ige Übereinstimmung -> gefunden
Welches Element löschen (0...11) : 5
2 Elemente gelöscht
Mset1 : 0 0 1 1 2 2 3 3 4 4 6 6 7 7 8 8 9 9 10 11
```

Die Container-Klassen »map« und »multimap«

Mit `map` bzw. `multimap` haben Sie wie schon bei `set` einen sortierten assoziativen Container – mit dem Unterschied, dass `map` bzw. `multimap` zwei Elemente auf-

nehmen kann (`map<Key, T>`) statt – wie `set` – nur eins. Das erste Element ist nach wie vor der Schlüssel, und das zweite Element ist gewöhnlich das Datenelement, mit dem der Schlüssel gefunden werden soll.

Der Unterschied zwischen `map` und `multimap` ist derselbe wie der zwischen `set` und `multiset`. Bei `map` darf ein Schlüssel (erstes Element) nur einmal vorhanden sein – `multimap` hingegeben erlaubt mehrere gleiche Schlüssel. Wie auch `set` bzw. `multiset` ist `map` bzw. `multimap` gewöhnlich über einen binären Baum realisiert (wenn auch der Standard dies nicht vorschreibt).

Auch hier gilt: Wenn die Schlüssel für den assoziativen Zugriff nicht sortiert sein müssen, liegen diese bei `map` bzw. `multimap` ebenfalls sortiert vor, was einen effizienteren Zugriff auf die einzelnen Elemente erlaubt.

Der Typ eines `map`- bzw. `multimap`-Elements ist `pair<const Key, T>` – `pair` wurde in Abschnitt 5.3.2, »Hilfsmittel (Hilfsstrukturen)«, beschrieben. Hier ist der Schlüssel immer konstant, damit er nicht mehr verändert werden kann. Wäre es möglich, den Schlüssel zu verändern, würde die ganze Sortierung im Container durcheinandergebracht.

Die Deklaration dieser Klasse hat folgende Syntax:

```
template<class Key,                    // Schlüssel
         class T,                      // Daten
         class Compare = less<Key> > // Vergleich
class map;
```

Bei der Sortierung `Compare` wird grundsätzlich der Operator `<` verwendet (standardmäßig ist dies `less<Key>`). Natürlich können Sie hierfür wieder eigene oder vordefinierte Funktionsobjekte einsetzen. Sortiert wird anhand des Schlüssels.

Die Klasse `map` stellt im Grunde dieselben Datentypen und Methoden wie `set` zur Verfügung. Das Prinzip ist eigentlich auch dasselbe, nur dass `map` zwei statt einen Wert verwendet. Trotzdem gibt es eine Besonderheit: `map` bzw. `multimap` stellt auch den Indexoperator zur Verfügung, um über den Index als Schlüssel auf die Daten zuzugreifen. Damit ist ein echtes assoziatives Array möglich. Dabei muss der Schlüssel keineswegs eine Zahl, sondern kann auch eine Zeichenkette sein:

```
plz["Hamburg"] = 20095;
```

Sollte bei einem Zugriff über den Indexoperator ein entsprechender Schlüssel nicht existieren, so wird über den Standardkonstruktor gleich ein Objekt erzeugt und in die Menge aufgenommen. Natürlich kann dies auch bedeuten, dass hierbei unbeabsichtigt neue Objekte erzeugt werden. Das kann man vermeiden, indem man überprüft, ob das Objekt existiert.

Neben den bereits gezeigten öffentlichen Datentypen (siehe Tabelle 5.4) stehen Ihnen für die Klasse `map` bzw. `multimap` folgende weitere Datentypen zur Verfügung:

Datentyp	Bedeutung
`pointer`	Zeiger auf `map`-Element
`const_pointer`	Zeiger auf `map`-Element, der nur lesend verwendet werden kann
`key_type`	Key
`value_type`	entspricht `pair<const Key, T>`
`mapped_type`	entspricht `T`
`key_compare`	Compare (Vorgabe `less<T>`)
`value_compare`	Klasse für Funktionsobjekte

Tabelle 5.38 Weitere Datentypen für die Container »set« und »multiset«

Abgesehen von den für alle Container vorhandenen Methoden, stehen dem Container `map` bzw. `multimap` noch folgende Methoden zur Verfügung (siehe Tabellen 5.39 bis 5.43):

Methoden	Bedeutung
`explicit map( const Compare& comp=Compare(), const Allocator&=Allocator() );`	Erzeugt ein `map` mit der Länge 0.
`template class<InputIterator> map( InputIterator first, InputIterator second, const Compare& comp=Compare(), const Allocator&=Allocator() );`	Erzeugt ein `map` mit der Länge 0 und fügt anschließend die Elemente `[first,last)` ein.
`map( const map< T, Compare, Allocator>& s );`	Erzeugt ein `map` als Kopie von `s`.
`~map();`	Zerstört ein `map` (gibt es frei).

Tabelle 5.39 Konstruktoren und Destruktor von »map« bzw. »multimap«

[»] Hinweis

Zwar wird hier vorwiegend die Syntax von `map` verwendet, die Syntax von `multimap` ist aber gleich, nur dass `multimap` statt `map` geschrieben wird und `multimap` mehrere gleiche Schlüssel erlaubt.

Methode	Bedeutung
`iterator erase(iterator pos);`	Löscht das Element mit der Position `pos`. Der zurückgegebene Iterator verweist auf den Nachfolger von `pos` oder auf `end()`, falls das letzte Element gelöscht wurde.
`iterator erase ( iterator first, iterator last );`	Löscht die Elemente im Bereich `[first,last)`. Der zurückgegebene Iterator verweist auf den Nachfolger von `last` oder auf `end()`, falls `last` nicht existiert.
`size_type erase ( const T& s );`	Löscht das Objekt `s` (falls vorhanden). Rückgabewert `1` = gelöscht; `0` = nicht gelöscht. Bei `multimap` werden alle Elemente `s` gelöscht.
`void clear();`	Löscht alle `map`-Elemente, so dass `map` die Länge 0 besitzt.

Tabelle 5.40 Methoden zum Löschen von Elementen

Methode	Bedeutung
`pair<iterator, bool> insert( const T& obj );`	Das Objekt `obj` wird eingefügt, falls es noch nicht existiert (gilt nicht für `multimap`). Der Iterator des zurückgegebenen Paares zeigt auf den eingefügten Schlüssel bzw. auf den schon vorhandenen Schlüssel mit demselben Wert wie `obj`. Der boolesche Wert zeigt, ob überhaupt ein Einfügen stattgefunden hat (es kann ja sein, dass der Schlüssel bereits existiert).
`iterator insert( iterator pos, const T& obj );`	Fügt das Objekt `obj` in `map` ein, falls `obj` noch nicht existiert (gilt nicht für `multimap`). Die Position `pos` ist lediglich ein Hinweis, ab wo `set` eingefügt werden soll (die Objekte werden ja sowieso sortiert).
`template <class InputIterator> void insert( InputIterator first, InputIterator last );`	Fügt die Elemente des Bereichs `[first, last)` eines anderen Containers in den aktuellen ein. Auch hierbei werden nur die Elemente eingefügt, die noch nicht vorhanden sind (gilt nicht für `multimap`).

Tabelle 5.41 Methoden zum Einfügen neuer Elemente

Methoden	Bedeutung
`iterator find( const T& s ) const;`	Sucht nach dem Objekt `s` in `map`. Der Rückgabewert ist die Position von `s` oder `end()`, falls `s` nicht vorhanden ist.

Tabelle 5.42 Suchoperationen für »map«

Methoden	Bedeutung
`iterator lower_bound( const T& s ) const;`	Gibt die Position des ersten Objekts in `map` zurück, das größer gleich `s` ist (auch als `const`-Version vorhanden).
`iterator upper_bound( const T& s ) const;`	Gibt die erste Position des Objekts in `map` zurück, das nicht kleiner als `s` ist (auch als `const`-Version vorhanden).
`pair<iterator, iterator> equal_range( const T& s ) const;`	Liefert die beiden Werte `upper_bound` und `lower_bound` als Paar.
`size_type count ( const T& s ) const;`	Gibt `1` zurück, wenn `s` im Set vorkommt, ansonsten `0`. Beim `multimap` gibt diese Methode die Anzahl zurück, die `s` in `multimap` vorhanden ist.

Tabelle 5.42 Suchoperationen für »map« (Forts.)

Methoden	Bedeutung
`void swap( map<T, Compare, Allocator>& s );`	Den Inhalt des map `*this` mit `s` vertauschen.
`map<T, Compare, Allocator>& operator=( const set<T, Compare, Allocator>& s );`	Zuweisung eines map `s` an das map `*this`
`key_compare key_comp() const`	Gibt eine Kopie des Vergleichsobjekts zurück, das bei der Erzeugung der map verwendet wurde.
`value_compare value_comp() const`	Gibt ein Funktionsobjekt zurück, das zum Vergleich von Objekten des Typs `value_type` (also Paaren) benutzt werden kann. Dieses Funktionsobjekt vergleicht zwei Paare auf der Basis ihrer Schlüssel und des Vergleichsobjekts, das bei der Erzeugung der map benutzt wurde.

Tabelle 5.43 Weitere Methoden

[»] **Hinweis**

In STL gibt es eine allgemeine Funktion `find()`, die für alle Container geeignet ist, und `map` bietet ebenfalls eine eigene Funktion `find()`. Die Elementfunktion von `map` ist hierbei auch wieder wesentlich schneller als die allgemeine.

Hierzu wieder ein einfaches Beispiel, das den Einsatz von `map` bzw. `multimap` in der Praxis demonstrieren soll:

```
// stl_map.cpp
#include <iostream>
#include <cstring>
#include <map>
using namespace std;
```

```
class A {
private:
   int ival;
   float fval;
public:
   // Konstruktor
   A( int i=0, float f=0.0f ): ival(i), fval(f) {}
   // Destruktor
   ~A( ) {}
   void anzeigen() const {
      cout << ival << " : " << fval << "\n";
   }
   int get_ival() const { return ival; }
   float get_fval() const { return fval; };
   // Überladener <-Operator
   friend bool operator <( const A& ival1, const A& ival2 ){
      if(ival1.ival < ival2.ival)
         return true;
      else
         return false;
   }
};

// Funktionsobjekt für C-String-Vergleiche
class less_str {
public:
   bool operator()(const char* s1, const char* s2) const {
      return strcmp(s1, s2) < 0;
   }
};

// Wir sparen uns hierbei Tipparbeit
typedef pair<const char*, const char*> kz_pair;
typedef pair<A, const char*> Ac_pair;

int main() {
   map<const char*, long, less_str> plz;
   map<A, const char*> KlasseA;
   multimap<const char*, const char*, less_str> kennzeichen;
   char ort[25];
   char kfz[5];

   plz["Mering"]  = 86415;
   plz["Bonn"]    = 53111;
   plz["Hamburg"] = 20095;
   KlasseA.insert(Ac_pair( A(1, 1.1f), "Erstes Beispiel"));
```

```
KlasseA.insert(Ac_pair( A(2, 2.2f), "Zweites Beispiel"));
KlasseA.insert(Ac_pair( A(3, 3.3f), "Drittes Beispiel"));

kennzeichen.insert(kz_pair("AIC", "Mering"));
kennzeichen.insert(kz_pair("AIC", "Aichach"));
kennzeichen.insert(kz_pair("AIC", "Friedberg"));
kennzeichen.insert(kz_pair("M", "Ismaning"));
kennzeichen.insert(kz_pair("M", "München"));
kennzeichen.insert(kz_pair("M", "Unterhaching"));
kennzeichen.insert(kz_pair("BN", "Bonn"));
kennzeichen.insert(kz_pair("HH", "Hamburg"));

cout << "Nach Postleitzahlen suchen\n";
cout << "Ortschaft eingeben: ";
cin >> ort;
map<const char*, long>::iterator iter = plz.find(ort);
if( iter != plz.end() )
    cout << "Postleitzahl zu " << ort << " : "
         << plz[ort] << "\n";
else
    cout << "Nichts gefunden zu " << ort << "\n";
cout << "Nach Autonummern suchen\n";
cout << "Autonummer eingeben: ";
cin >> kfz;
multimap<const char*, const char*>::iterator iter2 =
   kennzeichen.find(kfz);
if( iter2 != kennzeichen.end() ) {
    // mehrere Vorkommen?
    if( kennzeichen.count(kfz) > 1 ) {
       for( multimap<const char*, const char*>::iterator
            iter3 = kennzeichen.begin();
            iter3 != kennzeichen.end(); iter3++) {
          if( strcmp(iter3->first, kfz ) == 0 )
             cout << "Ortschaft zu " << kfz << " : "
                  << iter3->second << "\n";
       }
    }
    else  // nur ein Vorkommen
       cout << "Ortschaft zu " << kfz << " : "
            << iter2->second << "\n";
}
else
   cout << "Ortschaft zu " << kfz << " nicht gefunden\n";

// Alle Elemente von KlasseA ausgeben
for( map<A,const char*>::iterator iter = KlasseA.begin();
```

```
                iter != KlasseA.end(); iter++) {
      cout << iter->second << " : ";
      iter->first.anzeigen();
   }
   // Daten anhand eines Schlüssels löschen
   int cnt = kennzeichen.erase("AIC");
   if( cnt != 0 )
      cout << cnt << " Kennzeichendaten für AIC gelöscht\n";
    return 0;
}
```

Das Programm bei der Ausführung:

```
Nach Postleitzahlen suchen
Ortschaft eingeben: Mering
Postleitzahl zu Mering : 86415
Nach Autonummern suchen
Autonummer eingeben: M
Ortschaft zu M : Ismaning
Ortschaft zu M : München
Ortschaft zu M : Unterhaching
Erstes Beispiel : 1 : 1.1
Zweites Beispiel : 2 : 2.2
Drittes Beispiel : 3 : 3.3
3 Kennzeichendaten für AIC gelöscht
```

Weitere und zukünftige STL-Container-Klassen

Die hier aufgeführten Container-Klassen sind wohl die gängigsten von STL und alle im Standard aufgenommen. STL bietet aber noch eine Reihe weiterer interessanter Container, die teilweise offiziell im Standard aufgenommen wurden, sowie einige andere, die von vielen Compilern bereits unterstützt werden.

- Container-Klasse `slist` (sequentieller Container) – `slist` ist eine einfach verkettete Liste (*single list*), bei der jedes Element mit dem nächsten Element verkettet ist. Das bedeutet, dass die einzelnen Elemente nur vorwärts, aber niemals rückwärts (wie bei `list`) durchlaufen werden. Somit sind die Iteratoren für `slist` Forward-Iteratoren und keine (im Gegensatz zu `list`) bidirektionalen Iteratoren. Der Vorteil und der Grund der Implementierung von `slist` ist, dass einfach verkettete Listen kleiner sind (weniger Speicherplatz benötigen) und natürlich schneller als doppelt verkettete Listen (`list`). Sofern eines der beiden Kriterien für Ihr Projekt von Bedeutung ist, können Sie ja mal eine Version mit `slist` und eine mit `list` testen.
- Container-Klasse `bit_vector` (`vector<bool>`; sequentieller Container) – ein `bit_vector` ist nichts anderes als ein `vector<bool>` und hat dieselbe Schnittstelle wie ein `vector` mit Ausnahme des Datentyps `reference`, der für Mani-

pulationen an einzelnen Bits gedacht ist. Der Unterschied ist im Grunde nur, dass ein `bit_vector` auf Speicherplatzgröße optimiert ist. Ein `vector` benötigt mindestens ein Byte pro Element, bei einem `bit_vector` kann man dies auf ein Bit pro Element beschränken. Allerdings wird der Name `bit_vector` bei einem der nächsten Release von STL entfernt, stattdessen wird eine Spezialisierung von `vector<bool>` verwendet werden. Der Name `bit_vector` ist sowieso nur ein `typedef` für `vector<bool>`, der allerdings nicht im C++-Standard definiert ist. Zum Zweck der Rückwärts-Kompatibilität wird dieser Name natürlich bestehen bleiben.

- Container-Klasse `bitset` – der Container `bitset` ist wieder dem von `vector<bool>` (auch bekannt als `bit_vector`) recht ähnlich. Dieses Klassen-Template dient zum Abspeichern einzelner Bit-Folgen. Die Anzahl der Bits ist dabei nicht an die Größe eines Datentyps gebunden. So können Sie mit `bitset` praktisch 2 oder gar 100000 Bits in einer Folge abspeichern. Dennoch hat ein `bitset` gegenüber `vector<bool>` zwei entscheidende Unterschiede: Zum einen kann die Größe eines `bitset` nicht mehr nachträglich verändert werden. Die Anzahl Bits `N` muss als eine Integer-Konstante angegeben werden. Der zweite Unterschied ist, dass `bitset` kein sequentieller Container und somit auch kein STL-Container ist. `bitset` hat keine Iteratoren, und auch die Methoden sind nichts anderes als bitweise arithmetische Operatoren. Sofern Sie eine Bit-Manipulation in Ihrem Programm benötigen, ist `bitset` für Sie geeignet.
- Assoziative Hash-Container (kein C++-Standard) – die assoziativen STL-Container, die Sie hier kennengelernt haben, verwenden alle binäre Bäume, um die Elemente in einer sortierten Reihenfolge zu halten. Beim Einfügen und Löschen von Elementen kann dieses sortierte Verhalten ein Performance-Nachteil sein, wenn die Anzahl der Elemente sehr hoch ist. Für die Sortierung haben sich Hash-Algorithmen als effizienter erwiesen. Hierzu bieten die STL-Implementierungen die Container-Klassen `hash_set`, `hash_multiset`, `hash_map` und `hash_multimap` als Alternativen für `set`, `multiset`, `map` und `multimap` an. Allerdings dürfte noch einige Zeit vergehen, bis dies in den C++-Standard mit einfließt, wobei viele Compiler dieses Feature bereits anbieten.
- Container-Klassen `string` und `wstring` (sequentieller Container) – auch die Container-Klassen `string` und `wstring` (ähnlich wie `vector`) dienen zur sequentiellen Speicherung von Elementen – in diesem Fall aber zur Speicherung von Zeichenketten. `string` verwendet als Elementtyp `char` und dient als Ersatz für die fehleranfälligen C-Strings (ebenso wie `wstring` für *wide characters* als Ersatz für `wchar_t`). `string` macht den Umgang mit Zeichenketten zum Kinderspiel und bietet im Gegensatz zu `char*` eine sehr komfortable Handhabung, besonders was die Speicherverwaltung betrifft. Natürlich hat dies auch

seinen Preis: `string` benötigt mehr Rechenaufwand als die C-Strings. Auf die Container-Klasse `string` wird aber noch explizit in einem eigenen Abschnitt – siehe Abschnitt 7.1, »Die String-Bibliothek (string-Klasse)« – eingegangen.

5.3.6 Algorithmen

Neben den Containern bietet STL noch sehr viele Algorithmen an, die auch (und vor allem) mit den Container-Klassen zusammenarbeiten. Alle Algorithmen von STL im Detail zu erläutern, führt zu weit und ist auch gar nicht nötig. Sofern das Prinzip der Verwendung verstanden wurde, kann man einen Algorithmus von STL ohne großen Aufwand im Programm implementieren. Um die Algorithmen zu verwenden, muss die Header-Datei <*algorithm*> eingebunden werden. Der Namensraum sollte hierbei natürlich auf `std` gesetzt werden.

Die Beschreibung der einzelnen Algorithmen in diesem Abschnitt erfolgt möglichst sortiert nach Themen in verschiedenen Tabellen. Zu vielen Algorithmen soll auch ein kurzes Anwendungsbeispiel gezeigt werden. Auch wenn bei diesen Beispielen vorwiegend STL-Container oder einfache Klassen verwendet werden, soll hier noch erwähnt werden, dass sich die Algorithmen sehr wohl auch mit den einfachen Basisdatentypen vertragen.

Algorithmen mit Prädikat

Bei den folgenden Algorithmen gibt es viele Versionen, die ein Prädikat verwenden können. Ein Prädikat ist im Grunde ein Funktionsobjekt – siehe Abschnitt 5.3.2, »Hilfsmittel (Hilfsstrukturen)« –, das dem Algorithmus mitgegeben wird und das einen booleschen Wert zurückgibt. Dieses Funktionsobjekt wird verwendet, um zu überprüfen, ob ein Objekt eine bestimmte Bedingung erfüllt. Trifft dies zu, gibt dieses Funktionsobjekt `true` zurück – ansonsten `false`. Wird `true` zurückgegeben, findet der Algorithmus Anwendung auf das Objekt. Wichtig ist außerdem, dass ein Prädikat die Objekte nicht verändern darf.

Neben einfachen Prädikaten gibt es auch binäre Prädikate, die zwei Parameter verwenden. Näheres zur Anwendung von Prädikaten erfahren Sie im Folgenden.

Nicht verändernde Sequenz-Algorithmen

In diesem Abschnitt werden zunächst die Sequenz-Algorithmen beschrieben, die eine Sequenz aber nicht verändern. Diese Algorithmen ändern weder das Element selbst noch die Reihenfolge der Elemente. Gewöhnlich arbeiten diese Algorithmen (siehe Tabellen 5.44 bis 5.49) mit Input- und Forward-Iteratoren und sind auf alle sequentiellen Standard-Container anwendbar. Alle Algorithmen (mit Ausnahme des Algorithmus `search`) haben die Komplexität O(N) – N = Anzahl der Elemente.

Algorithmus	Beschreibung
`template <class InputIterator, class Function> Function for_each( InputIterator first, InputIterator last, Function f);`	Dieser Algorithmus ruft für jedes Element im Bereich `[first,last)` das Funktionsobjekt `f` auf. Ein Rückgabewert von `f` wird ignoriert.
`template <class InputIterator, class T> InputIterator find( InputIterator first, InputIterator last, const T& value );`	Sucht nach der ersten Position des Wertes `value` im Bereich `[first, last)` und gibt diese Position zurück. Bei erfolgloser Suche wird ein Zeiger auf `end()` zurückgegeben.
`template <class InputIterator, class Predicate> InputIterator find_if( InputIterator first, InputIterator last, Predicate pred);`	Sucht nach dem ersten Element im Bereich `[first, last)`, auf das die Bedingung `pred` zutrifft. Der Rückgabewert ist die Position des Elements. Bei erfolgloser Suche wird ein Zeiger auf `end()` zurückgegeben.

Tabelle 5.44 Algorithmen zur Suche nach Elementen

Hierzu ein Listing, das die Algorithmen `for_each()`, `find()` und `find_if()` in der Praxis demonstriert:

```
// stl_algo1.cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

class A {
private:
   int ival;
   float fval;
public:
   // Konstruktor
   A( int i=0, float f=0.0f ): ival(i), fval(f) {}
   // Destruktor
   ~A( ) {}
   void anzeigen() const {
      cout << ival << " : " << fval << "\n";
   }
   int get_ival() const { return ival; }
   float get_fval() const { return fval; };
   // Überladener <-Operator
   friend bool operator < (const A& ival1,const A& ival2) {
```

```
      if(ival1.ival < ival2.ival)
         return true;
      else
         return false;
   }
   // Überladener == Operator für find
   friend bool operator == (const A& ival1,const A& ival2) {
      if(ival1.ival == ival2.ival &&
         ival1.fval == ival2.fval     )
         return true;
      else
         return false;
   }
};

class biggerthan4 {
public:
   // alle Elemente größer gleich 4 zurückliefern
   bool operator()(int x) {
       return ( x >= 4 );
   }
};

class biggerthan4_0 {
public:
   // alle Elemente größer gleich 4.0 zurückliefern
   bool operator()(A x) {
       return ( x.get_fval() >= 4 );
   }
};

void ausgabeINT(int x);
void ausgabeA(A x);

int main() {
   // ein int-Vektor mit 5 Elementen
   vector<int> Vectorint(5);
   // ein Vektor der Klasse A mit 3 Elementen
   vector<A> VectorA(5);

   // int-Vektor mit Werten füllen
   for(size_t i = 0; i < Vectorint.size(); ++i)
      Vectorint[i] = i*i;
   // A-vector mit Werten füllen
   float f=1.1;
   for(size_t i=0; i < VectorA.size(); ++i) {
```

```
      VectorA[i] =  A(i, f);
      f*=2;
   }
   // for_each
   cout << "for_each (vector<int>)       :\n ";
   for_each(Vectorint.begin(), Vectorint.end(), ausgabeINT);
   cout << endl << "for_each (vector<A>) :\n";
   for_each(VectorA.begin(), VectorA.end(), ausgabeA);

   // find ...
   cout << "find (vector<int>)           :\n";
   vector<int>::const_iterator iter =
      find( Vectorint.begin(), Vectorint.end(), 4 );
   if( iter != Vectorint.end() ) {
      cout << "Element mit dem Wert " << *iter
           << " gefunden\n";
   }
   cout << "find (vector<A>)             :\n";
   vector<A>::const_iterator iter2 =
      find( VectorA.begin(), VectorA.end(), A(1, 2.2) );
   if( iter2 != VectorA.end() ) {
      cout << "Element mit den Werten "
           << iter2->get_ival() << " und "
           << iter2->get_fval()
           << " gefunden\n";
   }
   // find_if ... (und for_each)
   cout << "find_if (vector<int>)        :\n";
   vector<int>::iterator iter3 =
      find_if( Vectorint.begin(),
               Vectorint.end(), biggerthan4() );
   if( iter3 != Vectorint.end() ) {
      for_each(iter3, Vectorint.end(), ausgabeINT);
   }
   cout << endl;
   cout << "find_if (vector<A>)        :\n";
   vector<A>::iterator iter4 =
      find_if( VectorA.begin(),
               VectorA.end(), biggerthan4_0() );
   if( iter4 != VectorA.end() ) {
      for_each(iter4, VectorA.end(), ausgabeA);
   }
   cout << endl;
   return 0;
}
```

```
void ausgabeINT(int x) {
   cout << x << " ";
}

void ausgabeA(A x) {
   cout << x.get_ival() << " : " << x.get_fval() << "\n";
}
```

Das Programm bei der Ausführung:

```
for_each (vector<int>)       :
 0 1 4 9 16
for_each (vector<A>)         :
0 : 1.1
1 : 2.2
2 : 4.4
3 : 8.8
4 : 17.6
find (vector<int>)           :
Element mit dem Wert 4 gefunden
find (vector<A>)             :
Element mit den Werten 1 und 2.2 gefunden
find_if (vector<int>)        :
4 9 16
find_if (vector<A>)          :
2 : 4.4
3 : 8.8
4 : 17.6
```

Algorithmus	Beschreibung

`template<class ForwardIterator1,`
`class ForwardIterator2>`
`ForwardIterator1 find_first_of(`
`ForwardIterator1 first1,`
`ForwardIterator1 last1,`
`ForwardIterator2 first2,`
`ForwardIterator2 last2);`
	Damit suchen Sie im Bereich `[first1, last1)` nach dem ersten Aufkommen eines Elements aus der durch `[first2, last2)` angegebenen Teilfolge. Als Rückgabewert erhalten Sie die Position des gefundenen Elements oder einen Zeiger auf `end()` bei erfolgloser Suche.

`template<class ForwardIterator1,`
`class ForwardIterator2>`
`ForwardIterator1 find_first_of (`
`ForwardIterator1 first1,`
`ForwardIterator1 last1,`
`ForwardIterator2 first2,`
`ForwardIterator2 last2,`
`BinaryPredicate pred);`
	Damit suchen Sie im Bereich `[first1, last1)` nach dem ersten Aufkommen eines Elements aus der durch `[first2, last2)` angegebenen Teilfolge. Zudem muss hier noch die Bedingung `pred` zutreffen. Als Rückgabewert erhalten Sie die Position des gefundenen Elements oder einen Zeiger auf `end()` bei erfolgloser Suche.

Tabelle 5.45 Elemente aus einer Menge suchen

Algorithmus	Beschreibung

`template <class ForwardIterator>`
`ForwardIterator adjacent_find(`
`  ForwardIterator first,`
`  ForwardIterator last );`
	Der Algorithmus sucht im angegebenen Bereich `[first,last)` nach zwei gleichen benachbarten Elementen. Zurückgegeben wird die Position des ersten Auftretens der benachbarten Elemente oder ein Zeiger auf `end()`, falls die Suche erfolglos blieb.

`template <class ForwardIterator,`
`  class BinaryPredicate>`
`ForwardIterator adjacent_find(`
`  ForwardIterator first,`
`  ForwardIterator last,`
`  BinaryPredicate binary_pred );`
	Wie oben, nur sucht dieser Algorithmus nach zwei benachbarten Elementen im Bereich `[first,last)`, auf die die Bedingung `binary_pred` zutrifft. Zurückgegeben wird die Position des ersten Auftretens der benachbarten Elemente oder ein Zeiger auf `end()`, falls die Suche erfolglos blieb.

`template <class ForwardIterator,`
`  class Size, class T>`
`ForwardIterator search_n(`
`  ForwardIterator first,`
`  ForwardIterator last,`
`  Size count,`
`  const T& value );`
	Dieser Algorithmus sucht im Bereich `[first,last)` nach `count` benachbarten Elementen von `value`. Zurückgegeben wird die Position des ersten Auftretens der benachbarten Elemente oder ein Zeiger auf `end()`, falls die Suche erfolglos blieb.

`template <class ForwardIterator,`
`  class Size, class T,`
`  class BinaryPredicate>`
`ForwardIterator search_n(`
`  ForwardIterator first,`
`  ForwardIterator last,`
`  Size count,`
`  const T& value,`
`  BinaryPredicate binary_pred);`
	Dieser Algorithmus sucht im Bereich `[first,last)` nach `count` benachbarten Elementen von `value`, wobei für jedes Element der Vergleich `binary_pred` zutreffen muss. Zurückgegeben wird die Position des ersten Auftretens der benachbarten Elemente oder ein Zeiger auf `end()`, falls die Suche erfolglos blieb.

Tabelle 5.46 Suche nach gleichen aufeinanderfolgenden Elementen

Hierzu ein Beispiel, das die Algorithmen `adjacent_find()` und `search_n()` in der Praxis demonstriert:

```
// stl_algo2.cpp
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

class A {
private:
   int ival;
   float fval;
```

```
public:
   // Konstruktor
   A( int i=0, float f=0.0f ): ival(i), fval(f) {}
   // Destruktor
   ~A( ) {}
   void anzeigen() const {
      cout << ival << " : " << fval << "\n";
   }
   int get_ival() const { return ival; }
   float get_fval() const { return fval; };
   // Überladener <-Operator
   friend bool operator < (const A& ival1, const A& ival2) {
      if(ival1.ival < ival2.ival)
         return true;
      else
         return false;
   }
   // Überladener ==-Operator für find
   friend bool operator == (const A& ival1,const A& ival2) {
      if(ival1.ival == ival2.ival &&
         ival1.fval == ival2.fval     )
         return true;
      else
         return false;
   }
};

// Gibt true zurück, wenn das Nachbarelement den
// doppelten Wert hat
bool verdoppelt ( int elem1, int elem2 ) {
   return elem1 *2 == elem2;
}

int main() {
   vector<int> Vector1;
   vector<A> VectorA;

   for( int i = 0; i <= 5; i++ ) {
      Vector1.push_back( 3 * i );
   }
   for( int i = 0; i <= 3; i++ ) {
      Vector1.push_back( 111 );
   }
   for( int i = 0; i <= 3; i++ ) {
```

```
   Vector1.push_back( 9 * i );
}
cout << "Vector1 = (";
vector<int>::const_iterator iter;
for( iter = Vector1.begin();
     iter != Vector1.end(); iter++ )
   cout << *iter << ", ";
cout << ")\n";

// Suche in Vector1 nach dem ersten Auftreten
// von dreimal 111
vector<int>::iterator iter2 =
   search_n (Vector1.begin(), Vector1.end(), 3, 111 );
if( iter2 != Vector1.end() )
   cout << "Es sind drei aufeinanderfolgende "
        << "Elemente mit dem Wert 111 vorhanden\n"
        << "Die Position des Auftretens lautet: "
        << iter2-Vector1.begin() << endl;
else
   cout << "Drei aufeinanderfolgende Elemente"
        << " mit dem Wert 111 gibt es nicht\n";

vector <int>::iterator iter3 =
   adjacent_find ( Vector1.begin( ), Vector1.end( ),
                   verdoppelt );
if ( iter3 != Vector1.end( ) )
   cout << "Es wurde ein Element gefunden, "
        << "dessen Nachbar den doppelten Wert hat\n"
        << "Die Position des Auftretens lautet : "
        << iter3 - Vector1.begin()
        << "\nDie Werte sind " << *iter3 << " und "
        << *(iter3++) << endl;

for( int i = 0; i <= 5; i++ ) {
   VectorA.push_back( A(3 * i, 1.1f ) );
}
for( int i = 0; i <= 3; i++ ) {
   VectorA.push_back( A(9 * i, 2.2f) );
}
for( int i = 0; i <= 3; i++ ) {
   VectorA.push_back( A(111, 111.111) );
}
cout << "VectorA = (";
vector<A>::const_iterator iterA;
```

```
   for( iterA = VectorA.begin();
        iterA != VectorA.end(); iterA++ )
      cout << "(" << iterA->get_ival() << ", "
           << iterA->get_fval() << "),";
   cout << ")\n";

   // Suche in VectorA nach dem ersten Auftreten
   // von dreimal (111, 111.111)
   vector<A>::iterator iterA2 =
      search_n ( VectorA.begin(), VectorA.end(),
                 3, A(111, 111.111) );
   if( iterA2 != VectorA.end() )
      cout << "Es sind drei aufeinanderfolgende "
           << "Elemente mit (111,111.111) vorhanden\n"
           << "Die Position des Auftretens lautet: "
           << iterA2-VectorA.begin() << endl;
   else
      cout << "Drei aufeinanderfolgende Elemente mit "
           << "dem Wert (111, 111.111) gibt es nicht\n";
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector1=(0,3,6,9,12,15,111,111,111,111,0,9,18,27,)
Es sind drei aufeinanderfolgende
Elemente mit dem Wert 111 vorhanden
Die Position des Auftretens lautet: 6

Es wurde ein Element gefunden,
dessen Nachbar den doppelten Wert hat
Die Position des Auftretens lautet : 2
Die Werte sind 6 und 3

VectorA=((0,1.1),(3,1.1),(6,1.1),(9,1.1),
         (12,1.1),(15,1.1),(0,2.2),(9,2.2),
         (18, 2.2),(27, 2.2),(111, 111.111),
         (111,111.111),(111,111.111),(111,111.111))
Es sind drei aufeinanderfolgende
Elemente mit dem Wert (111, 111.111) vorhanden
Die Position des Auftretens lautet: 10
```

Algorithmus	Beschreibung
`template <class InputIterator1, class InputIterator2> pair<InputIterator1, InputIterator2> mismatch( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);`	Dieser Algorithmus sucht im Bereich `[first1,last1)` und `[first2,first2+[last1-first1))` nach den ersten beiden Elementen, die unterschiedlich sind. Der Algorithmus gibt ein Paar von Iteratoren zurück, die auf die erste Stelle der Nichtübereinstimmung in den jeweiligen korrespondierenden Containern zeigen. Falls beide Container übereinstimmen, ist der erste Iterator des zurückgegebenen Paares gleich `last1`.
`template <class InputIterator1, class InputIterator2, class BinaryPredicate> pair<InputIterator1, InputIterator2> mismatch( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, BinaryPredicate binary_pred);`	Dito, nur muss zusätzlich noch die Bedingung `binary_pred` zutreffen.

Tabelle 5.47 Suche nach unterschiedlichen Elementen

Auch hier soll ein Listing zur Demonstration verwendet werden. Dabei soll sowohl die Standardversion als auch die überladene Version mit dem binären Prädikat zum Einsatz kommen:

```
// stl_algo3.cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
using namespace std;

// Gibt true zurück, wenn das Nachbarelement den
// doppelten Wert hat
bool verdoppelt ( int elem1, int elem2 ) {
   return elem1 *2 == elem2;
}

int main() {
   vector <int> Vector1, Vector2;
   list <int> List1;
   vector <int>::iterator Viter1, Viter2;
   list <int>::iterator Liter1, Liter2;

   for (int i = 0 ; i <= 5 ; i++ ) {
```

```
   Vector1.push_back( 5 * i );
}
for ( int i = 0 ; i <= 7 ; i++ ) {
   List1.push_back( 5 * i );
}
for ( int i = 0 ; i <= 5 ; i++ ) {
   Vector2.push_back( 10 * i );
}
cout << "Vector Vector1 = ( " ;
for ( Viter1 = Vector1.begin( ) ;
      Viter1 != Vector1.end( ) ; Viter1++ )
   cout << *Viter1 << " ";
cout << ")" << endl;

cout << "List List1 = ( " ;
for ( Liter1 = List1.begin( ) ;
      Liter1!= List1.end( ) ; Liter1++ )
   cout << *Liter1 << " ";
cout << ")" << endl;

cout << "Vector Vector2 = ( " ;
for ( Viter2 = Vector2.begin( ) ;
      Viter2 != Vector2.end( ) ; Viter2++ )
   cout << *Viter2 << " ";
   cout << ")" << endl;

// Überprüfe Vector1 und List1
pair< vector <int>::iterator,
      list <int>::iterator> results1;
results1 = mismatch ( Vector1.begin( ),
                      Vector1.end( ), List1.begin( ));

if ( results1.first == Vector1.end( ) )
   cout << "Beide Bereiche (Vector1 und List1)"
        << " unterscheiden sich nicht" << endl;
else
   cout << "Der erste Unterschied ist aufgetreten bei "
        << *results1.first << " und " << *results1.second
        << endl;

// Überprüfe Vector1 und Vector2 mit dem
// binären Prädikat verdoppelt
pair< vector <int>::iterator,
      vector <int>::iterator> results2;
results2 = mismatch ( Vector1.begin( ),
```

```
                        Vector1.end( ),
                        Vector2.begin( ), verdoppelt );

if ( results2.first == Vector1.end( ) )
   cout << "Beide Bereiche (Vector1 und Vector2) "
        << "unterscheiden sich nicht mit dem binären"
        << "Prädikat" << endl;
else
   cout << "Der erste Unterschied trat auf an Pos. "
        << *results2.first << " und " << *results2.second
        << endl;

// Das gleich nochmals, nur wird Vector2 verändert
Vector2[2] = 111;

cout << "Vector Vector1 = ( " ;
for ( Viter1 = Vector1.begin( ) ;
      Viter1 != Vector1.end( ) ; Viter1++ )
   cout << *Viter1 << " ";
cout << ")" << endl;

cout << "Vector Vector2 = ( " ;
for ( Viter2 = Vector2.begin( ) ;
      Viter2 != Vector2.end( ) ; Viter2++ )
   cout << *Viter2 << " ";
   cout << ")" << endl;

// Überprüfe Vector1 und Vector2 nochmals mit dem
// binären Prädikat verdoppelt
pair< vector <int>::iterator,
      vector <int>::iterator> results3;
results3 = mismatch ( Vector1.begin( ),
                        Vector1.end( ),
                        Vector2.begin( ), verdoppelt );

if ( results3.first == Vector1.end( ) )
   cout << "Beide Bereiche (Vector1 und Vector2) "
        << "unterscheiden sich nicht mit dem binären"
        << "Prädikat" << endl;
else
   cout << "Der erste Unterschied trat auf an Pos. "
        << *results3.first << " (Vector1) und "
        << *results3.second
        << " (Vector2)" << endl;
```

```
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector Vector1 = ( 0 5 10 15 20 25 )
List List1 = ( 0 5 10 15 20 25 30 35 )
Vector Vector2 = ( 0 10 20 30 40 50 )

Beide Bereiche (Vector1 und List1) unterscheiden sich nicht
Beide Bereiche (Vector1 und Vector2) unterscheiden sich
nicht mit dem binären Prädikat

Vector Vector1 = ( 0 5 10 15 20 25 )
Vector Vector2 = ( 0 10 111 30 40 50 )
Der erste Unterschied ist aufgetreten an Pos. 10 (Vector1)
und 111 (Vector2)
```

Algorithmus	Beschreibung

`template<class ForwardIterator1,`
`  class ForwardIterator2>`
`ForwardIterator1 search(`
`  ForwardIterator1 first1,`
`  ForwardIterator1 last1,`
`  ForwardIterator2 first2,`
`  ForwardIterator2 last2 );`
	Dieser Algorithmus sucht im Bereich `[first1,last1)` nach dem *ersten* Vorkommen der in `[first2, last2)` angegebenen Teilfolge. Als Rückgabewert erhält man die Position des ersten Elements der gefundenen Teilfolge oder `last1` bei erfolgloser Suche.

`template<class ForwardIterator1,`
`  class ForwardIterator2,`
`  class BinaryPredicate>`
`ForwardIterator1 search(`
`  ForwardIterator1 first1,`
`  ForwardIterator1 last1,`
`  ForwardIterator2 first2,`
`  ForwardIterator2 last2,`
`  BinaryPredicate binary_pred );`
	Dito, nur muss noch die Bedingung `binary_pred` zutreffen (`true`).

`template<class ForwardIterator1,`
`  class ForwardIterator2>`
`ForwardIterator1 find_end(`
`  ForwardIterator1 first1,`
`  ForwardIterator1 last1,`
`  ForwardIterator2 first2,`
`  ForwardIterator2 last2);`
	Dieser Algorithmus sucht im Bereich `[first1,last1)` nach dem *letzten* Vorkommen der in `[first2, last2)` angegebenen Teilfolge. Als Rückgabewert erhält man die Position des ersten Elements der gefundenen Teilfolge oder `last1` bei erfolgloser Suche.

Tabelle 5.48 Algorithmen zur Suche nach Teilfolgen

Algorithmus	Beschreibung

`template<class ForwardIterator1,`
`class ForwardIterator2>`
`ForwardIterator1 find_end(`
`ForwardIterator1 first1,`
`ForwardIterator1 last1,`
`ForwardIterator2 first2,`
`ForwardIterator2 last2,`
`BinaryPredicate pred );`
	Dito, nur muss noch die Bedingung `binary_pred` zutreffen (`true`).

`template <class InputIterator1,`
`class InputIterator2>`
`bool equal(`
`InputIterator1 first1,`
`InputIterator1 last1,`
`InputIterator2 first2);`
	Hier wird überprüft, ob die Elemente in den Bereichen `[first1,last1)` und `[first2, first2+(last1-first1))` gleich sind – dann wird `true`, ansonsten `false` zurückgegeben.

`template <class InputIterator1,`
`class InputIterator2,`
`class BinaryPredicate>`
`bool equal(`
`InputIterator1 first1,`
`InputIterator1 last1,`
`InputIterator2 first2,`
`BinaryPredicate binary_pred);`
	Dito, nur muss noch die Bedingung `binary_pred` zutreffen (`true`).

Tabelle 5.48 Algorithmen zur Suche nach Teilfolgen (Forts.)

Das folgende Beispiel demonstriert die beiden Algorithmen `search()` und `equal()` in der Praxis:

```
// stl_algo4.cpp
#include <iostream>
#include <algorithm>
#include <vector>
#include <list>
using namespace std;

int main() {
   vector <int> Vector1;
   list <int> List1;
   vector <int>::iterator Viter1;
   list <int>::iterator Liter1;
   bool test;

   for (int i = 1 ; i <= 7 ; i++ ) {
      Vector1.push_back( 5 * i );
   }
   for ( int i = 3 ; i <= 6 ; i++ ) {
```

```
   List1.push_back( 5 * i );
}
cout << "Vector Vector1 = ( " ;
for ( Viter1 = Vector1.begin( ) ;
      Viter1 != Vector1.end( ) ; Viter1++ )
   cout << *Viter1 << " ";
cout << ")" << endl;

cout << "List List1 = ( " ;
for ( Liter1 = List1.begin( ) ;
      Liter1!= List1.end( ) ; Liter1++ )
   cout << *Liter1 << " ";
cout << ")" << endl;

// Suche in Vector1 und nach einer Teilfolge von List1
vector <int>::iterator results1;
results1 = search ( Vector1.begin( ), Vector1.end( ),
                    List1.begin( ), List1.end() );

if ( results1 == Vector1.end( ) )
   cout << "Suche nach Teilfolge erfolglos" << endl;
else
   cout << "Teilfolge List1 in Vector1 gefunden an Pos. "
        << (results1-Vector1.begin()) << " (Wert: "
        << *results1 << ")" << endl;

// Vector1 und List1 auf Gleichheit überprüfen
test = equal( Vector1.begin(), Vector1.end(),
              List1.begin() );
if( test == true )
   cout << "Die angegebenen Bereiche sind gleich"
        << endl;
else
   cout << "Die angegebenen Bereiche sind ungleich"
        << endl;

// Liste löschen
List1.clear();
// Elemente von Vector1 in List1 kopieren
for ( Viter1 = Vector1.begin( ) ;
      Viter1 != Vector1.end( ) ; Viter1++ )
   List1.push_back(*Viter1);

// Nochmals auf Gleichheit überprüfen
test = equal( Vector1.begin(), Vector1.end(),
              List1.begin() );
```

```
   if( test == true )
      cout << "Die angegebenen Bereiche sind gleich"
           << endl;
   else
      cout << "Die angegebenen Bereiche sind ungleich"
           << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector Vector1 = ( 5 10 15 20 25 30 35 )
List List1 = ( 15 20 25 30 )
Teilfolge List1 in Vector1 gefunden an Pos. 2 (Wert: 15)
Die angegebenen Bereiche sind ungleich
Die angegebenen Bereiche sind gleich
```

Algorithmus	Beschreibung
`template <class InputIterator, class T> iterator_traits< InputIterator>::difference_type count( InputIterator first, InputIterator last, const T& value );`	Dieser Algorithmus liefert die Anzahl der Elemente mit dem Wert `value` zurück, die im Bereich `[first,last)` vorkommen.
`template <class InputIterator, class Predicate> iterator_traits< InputIterator>::difference_type count_if( InputIterator first, InputIterator last, Predicate pred);`	Hier wird die Anzahl der Elemente zurückgegeben, die im Bereich `[first,last)` vorkommen und auf die die Bedingung `pred` zutrifft.

Tabelle 5.49 Eine Anzahl von Elementen ermitteln

Hier folgt nun ein Beispiel zum Algorithmus `count_if()`:

```
// stl_algo5.cpp
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

bool bigger30 ( int value ) {
   return (value > 30);
}
```

```
int main( ) {
   vector <int> Vector1;
   vector <int>::iterator Viter;

   for(int i = 5; i < 50; i+=5 )
      Vector1.push_back( i );

   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   int res;
   res = count_if( Vector1.begin( ), Vector1.end( ),
                   bigger30 );
   cout << res
        << " Elemente in Vector1 sind größer als 30\n";
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector1 = ( 5 10 15 20 25 30 35 40 45 )
3 Elemente in Vector1 sind größer als 30
```

Verändernde Sequenz-Algorithmen

Algorithmus	Beschreibung
`template <class InputIterator, class OutputIterator> OutputIterator` **`copy`**`( InputIterator first, InputIterator last, OutputIterator result );`	Damit wird aus dem Bereich `[first,last)` in den Bereich `[result, result+(last-first))` kopiert, angefangen bei der Position `first` bis zur Position `last-1`. Dabei darf der Iterator `result` nicht im Bereich `[first,last)` liegen. Rückgabewert: `result+last-first`
`template < class BidirectionalIterator1, class BidirectionalIterator2> BidirectionalIterator2` **`copy_backward`**`( BidirectionalIterator1 first, BidirectionalIterator1 last, BidirectionalIterator2 result);`	Dieser Algorithmus wird im Gegensatz zu `copy()` verwendet, wenn Ziel- und Quellbereich sich so überlappen, dass der Anfang des Zielbereichs im Quellbereich liegt. `result` muss anfangs auf das Ende des Zielbereichs zeigen. Rückgabewert: `result-(last-first)`

Tabelle 5.50 Kopierende Algorithmen

Algorithmus	Beschreibung

`template <`
`  class BidirectionalIterator,`
`  class OutputIterator>`
`OutputIterator`
`  reverse_copy(`
`  BidirectionalIterator first,`
`  BidirectionalIterator last,`
`  OutputIterator result );`
	Hiermit werden die Elemente im Bereich `[first,last)` in den Bereich `[result, result + (last – first))` in umgekehrter Reihenfolge kopiert. `first` wird hiermit praktisch in `result + (last-first-1)` kopiert. Dabei dürfen sich die Bereiche `[first,last)` und `[result, result + (last – first))` überlappen. Rückgabewert: `result+(last-first)`

`template<class InputIterator,`
`  class Size,`
`  class OutputIterator>`
`OutputIterator copy_n(`
`  InputIterator first,`
`  Size count,`
`  OutputIterator result );`
	Kopiert `n` Elemente aus dem Bereich `[first, first+count)` in den Bereich `[result, result+count)`. Dieser Algorithmus ist allerdings noch nicht im Standard aufgenommen worden, aber bereits bei vielen Compilern vorhanden.

Tabelle 5.50 Kopierende Algorithmen (Forts.)

[»] Hinweis

Sicherlich stellt sich die Frage, warum es keinen Algorithmus gibt, der nur dann kopiert, wenn eine bestimmte Bedingung erfüllt ist – eine Art `copy_if()` eben. Einen solchen Algorithmus gibt es in der Tat, doch ist er nicht im Standard aufgenommen worden und befindet sich auch nicht in der STL-Bibliothek, sondern in der Boost-Bibliothek.

Hierzu ein Beispiel, das die beiden Algorithmen `copy()` und `reverse_copy` in der Praxis zeigen soll:

```
// stl_algo6.cpp
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
using namespace std;

int main( ) {
   vector<int> Vector1;
   vector<int> Vector2;
   list<int> List1;
   list<int> List2;
   vector<int>::iterator Viter;
   list<int>::iterator Liter;

   for(int i = 5; i < 50; i+=5 )
      Vector1.push_back( i );
   for(int i = 50; i < 100; i+=5 )
```

```
   List1.push_back( i );

cout << "Vector1 = ( " ;
for ( Viter = Vector1.begin( ) ;
      Viter != Vector1.end( ) ; Viter++ )
   cout << *Viter << " ";
cout << ")" << endl;

cout << "List1 = ( " ;
for ( Liter = List1.begin( ) ;
      Liter != List1.end( ) ; Liter++ )
   cout << *Liter << " ";
cout << ")" << endl;

// Platz machen für neue Elemente
Vector2.resize(Vector1.size());
// Komplettes Vector1 in Vector2 kopieren
copy( Vector1.begin(), Vector1.end(), Vector2.begin() );

cout << "Vector2 = ( " ;
for ( Viter = Vector2.begin( ) ;
      Viter != Vector2.end( ) ; Viter++ )
   cout << *Viter << " ";
cout << ")" << endl;

// Platz für List2 machen
List2.resize(Vector2.size());
// Kompletten Vector2 rückwärts in List2 kopieren
reverse_copy( Vector2.begin(), Vector2.end(),
              List2.begin());
cout << "List2 = ( " ;
for ( Liter = List2.begin( ) ;
      Liter != List2.end( ) ; Liter++ )
   cout << *Liter << " ";
cout << ")" << endl;

// Die ersten drei Elemente von Vector1
// am Anfang von List1 kopieren
copy( Vector1.begin(), Vector1.begin()+3, List1.begin());
cout << "List1 = ( " ;
for ( Liter = List1.begin( ) ;
      Liter != List1.end( ) ; Liter++ )
   cout << *Liter << " ";
cout << ")" << endl;
```

```
   // Die Elemente 4 bis 7 aus Vector1
   // an die zweite Pos. von Vector2 kopieren
   copy(Vector1.begin()+4, Vector1.begin()+7,
        Vector2.begin()+2 );
   cout << "Vector2 = ( " ;
   for ( Viter = Vector2.begin( ) ;
         Viter != Vector2.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector1 = ( 5 10 15 20 25 30 35 40 45 )
List1 = ( 50 55 60 65 70 75 80 85 90 95 )
Vector2 = ( 5 10 15 20 25 30 35 40 45 )
List2 = ( 45 40 35 30 25 20 15 10 5 )
List1 = ( 5 10 15 65 70 75 80 85 90 95 )
Vector2 = ( 5 10 25 30 35 30 35 40 45 )
```

Algorithmus	Beschreibung
`template <class T>` `void swap(T& a, T& b);`	Vertauscht die beiden Objekte `a` und `b`.

`template <`
`  class ForwardIterator1,`
`  class ForwardIterator2>`
`void iter_swap(`
`  ForwardIterator1 a,`
`  ForwardIterator2 b );`
	Vertauscht die beiden Elemente in den Positionen `a` und `b`.

`template <`
`  class ForwardIterator1,`
`  class ForwardIterator2>`
`ForwardIterator2 swap_ranges(`
`  ForwardIterator1 first1,`
`  ForwardIterator1 last1,`
`  ForwardIterator2 first2 );`
	Vertauscht die beiden Elemente im Bereich `[first1,last1)` und `[first2, first2, +(last1 – first1)`, angefangen beim Element `*first1` und `*first2`.
	Rückgabewert: `first2+(last1-first1)`

`template <`
`  class BidirectionalIterator>`
`void reverse(`
`  BidirectionalIterator first,`
`  BidirectionalIterator last);`
	Vertauscht das erste Element `*first` mit dem letzten Element `*last`, dann das zweite Element mit dem vorletzten Element usw. bis zum mittleren Element.

Tabelle 5.51 Austauschen von Elementen (bzw. Teilbereichen)

Hier folgt nun ein Beispiel zu den beiden Algorithmen `swap_ranges()` und `reverse()`:

```
// stl_algo7.cpp
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
using namespace std;

int main( ) {
   vector<int> Vector1;
   list<int> List1;
   vector<int>::iterator Viter;
   list<int>::iterator Liter;

   for(int i = 5; i < 50; i+=5 )
      Vector1.push_back( i );
   for(int i = 50; i < 100; i+=5 )
      List1.push_back( i );

   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   cout << "List1 = ( " ;
   for ( Liter = List1.begin( ) ;
         Liter != List1.end( ) ; Liter++ )
      cout << *Liter << " ";
   cout << ")" << endl << endl;

   // Alle Elemente von Vector1 mit List1 austauschen
   swap_ranges( Vector1.begin(), Vector1.end(),
                List1.begin());
   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   cout << "List1 = ( " ;
   for ( Liter = List1.begin( ) ;
         Liter != List1.end( ) ; Liter++ )
```

```
      cout << *Liter << " ";
   cout << ")" << endl;

   // reverses Tauschen bis zum mittleren Element
   reverse( Vector1.begin(), Vector1.end() );
   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector1 = ( 5 10 15 20 25 30 35 40 45 )
List1 = ( 50 55 60 65 70 75 80 85 90 95 )

Vector1 = ( 50 55 60 65 70 75 80 85 90 )
List1 = ( 5 10 15 20 25 30 35 40 45 95 )
Vector1 = ( 90 85 80 75 70 65 60 55 50 )
```

Algorithmus	Beschreibung

`template <class InputIterator,`
`class OutputIterator,`
`class UnaryOperation>`
`OutputIterator transform(`
`InputIterator first,`
`InputIterator last,`
`OutputIterator result,`
`UnaryOperation op);`
	Hiermit wird auf jedes Element des Bereichs `[first,last)` die Operation `op` angewendet und das Ergebnis in den mit `result` beginnenden Bereich kopiert. `result` darf hier identisch mit dem Bereich `first` sein, allerdings werden dann die Elemente durch die transformierten ersetzt. Rückgabewert: `result+(last1-first1)`

`template <class InputIterator1,`
`class InputIterator2,`
`class OutputIterator,`
`class BinaryOperation>`
`OutputIterator transform(`
`InputIterator1 first1,`
`InputIterator1 last1,`
`InputIterator2 first2,`
`OutputIterator result,`
`BinaryOperation bin_op);`
	In dieser Form werden zwei Bereiche verwendet. Der erste Bereich ist`[first1, last1)` und der zweite `[first2, first2+last1-first1)` – somit ist der erste Bereich genauso groß wie der zweite. Die Operation `bin_op` nimmt jeweils aus jedem dieser Bereiche ein Element und legt das entsprechende Ergebnis in `result` ab. `result` darf identisch mit `first1` oder `first2` sein, wobei dann die Elemente durch die transformierten ersetzt werden. Rückgabewert: `result+(last1-first1)`

Tabelle 5.52 Elemente transformieren

Hierzu ein Listing, das die Funktion `transform()` in der Praxis demonstriert:

```
// stl_algo8.cpp
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
#include <iterator>   // ostream_iterator
using namespace std;

int main( ) {
   vector<int> Vector1;
   list<int> List1;
   vector<int>::iterator Viter;
   list<int>::iterator Liter;

   for(int i = 5; i < 25; i+=5 )
      Vector1.push_back( i );
   for(int i = 50; i < 70; i+=5 )
      List1.push_back( i );

   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   cout << "List1 = ( " ;
   for ( Liter = List1.begin( ) ;
         Liter != List1.end( ) ; Liter++ )
      cout << *Liter << " ";
   cout << ")" << endl;

   // Gibt jeweils die Differenz von zwei
   // Elementen (Vector1, List1) aus
   cout << "Differenz: ";
   transform (Vector1.begin(), Vector1.end(),
              List1.begin(),ostream_iterator<int>(cout," "),
              minus<int>());
   cout << endl;

   // Aus allen Elementen in Vector1 einen
   // negativen Wert machen
   transform (Vector1.begin(), Vector1.end(),
              Vector1.begin(), negate<int>());
```

```
   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   // Transformiert Elemente von Vector1 in
   // List1 mit dem 2fachen Wert
       transform (Vector1.begin(), Vector1.end(),
                  back_inserter(List1),
                  bind2nd(multiplies<int>(),2));

   cout << "List1 = ( " ;
   for ( Liter = List1.begin( ) ;
         Liter != List1.end( ) ; Liter++ )
      cout << *Liter << " ";
   cout << ")" << endl;

   cout << "List1 = ( " ;
   // Gibt List1 mit negativem Wert und rückwärts aus
   transform (List1.rbegin(), List1.rend(),
              ostream_iterator<int>(cout," "),
              negate<int>());
   cout << ")" << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector1 = ( 5 10 15 20 )
List1 = ( 50 55 60 65 )
Differenz: -45 -45 -45 -45
Vector1 = ( -5 -10 -15 -20 )
List1 = ( 50 55 60 65 -10 -20 -30 -40 )
List1 = ( 40 30 20 10 -65 -60 -55 -50 )
```

Algorithmus	Beschreibung

`template <class ForwardIterator, class T>`
`void replace(`
`  ForwardIterator first,`
`  ForwardIterator last,`
`  const T& old_value,`
`  const T& new_value);`
	Ersetzt die Elemente mit dem Wert `old_value` im Bereich `[first,last)` durch den Wert `new_val`.

Tabelle 5.53 Ersetzen von Elementen und Varianten

Algorithmus	Beschreibung
`template <class ForwardIterator, class Predicate, class T> void replace_if( ForwardIterator first, ForwardIterator last, Predicate pred, const T& new_value);`	Ersetzt die Elemente aus dem Bereich `[first,last)`, auf den die Bedingung `pred` zutrifft, durch den Wert `new_value`.
`template <class InputIterator, class OutputIterator, class T> OutputIterator replace_copy( InputIterator first, InputIterator last, OutputIterator result, const T& old_value, const T& new_value);`	Kopiert die Elemente aus dem Bereich `[first,last)` in den Bereich, der bei `result` beginnt. Ist der Wert ein Element aus dem Bereich `old_value`, wird dieser durch `new_value` ersetzt. Rückgabewert: `result+(last-first)`
`template <class Iterator, class OutputIterator, class Predicate, class T> OutputIterator replace_copy_if( Iterator first, Iterator last, OutputIterator result, Predicate pred, const T& new_value);`	Kopiert die Elemente aus dem Bereich `[first,last)` in den Bereich, der bei `result` beginnt. Trifft die Bedingung `pred` auf das Element zu, wird das Element durch `new_value` ersetzt. Rückgabewert: `result+(last-first)`

Tabelle 5.53 Ersetzen von Elementen und Varianten (Forts.)

Hierzu ein Beispiel, das die Funktionen `replace()`, `replace_if()`, `replace_copy()` und `replace_copy_if()` in der Praxis zeigen soll:

```
// stl_algo9.cpp
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
#include <iterator>   // ostream_iterator
using namespace std;

int main( ) {
   vector<int> Vector1;
   list<int> List1;
   vector<int>::iterator Viter;
   list<int>::iterator Liter;
```

```
for(int i = 5; i < 25; i+=5 )
   Vector1.push_back( i );
for(int i = 50; i < 70; i+=5 )
   List1.push_back( i );

cout << "Vector1 = ( " ;
for ( Viter = Vector1.begin( ) ;
      Viter != Vector1.end( ) ; Viter++ )
   cout << *Viter << " ";
cout << ")" << endl;

cout << "List1 = ( " ;
for ( Liter = List1.begin( ) ;
      Liter != List1.end( ) ; Liter++ )
   cout << *Liter << " ";
cout << ")" << endl;

// Den Wert 10 durch den Wert 111 ersetzen
replace( Vector1.begin(), Vector1.end(), 10, 111 );
cout << "Vector1 = ( " ;
for ( Viter = Vector1.begin( ) ;
      Viter != Vector1.end( ) ; Viter++ )
   cout << *Viter << " ";
cout << ")" << endl;

// Alle Elemente, die kleiner als 60 sind,
// durch 100 ersetzen

replace_if( List1.begin(), List1.end(),
            bind2nd(less<int>(),60), 100 );
cout << "List1 = ( " ;
for ( Liter = List1.begin( ) ;
      Liter != List1.end( ) ; Liter++ )
   cout << *Liter << " ";
cout << ")" << endl;

cout << "List1 = ( " ;
// Gibt statt des Wertes 100 den ersetzten Wert 666 aus
replace_copy( List1.begin(), List1.end(),
              ostream_iterator<int>(cout, " "),
              100, 666 );
cout << ")" << endl;

cout << "Vector1 = ( " ;
// Gibt alle Elemente mit dem Wert kleiner
```

```
   // als 20 aus und ersetzt diese durch den Wert 0
   replace_copy_if( Vector1.begin(), Vector1.end(),
                    ostream_iterator<int>(cout, " "),
                    bind2nd(less<int>(), 20), 0 );
   cout << ")" << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector1 = ( 5 10 15 20 )
List1 = ( 50 55 60 65 )
Vector1 = ( 5 111 15 20 )
List1 = ( 100 100 60 65 )
List1 = ( 666 666 60 65 )
Vector1 = ( 0 111 0 20 )
```

Algorithmus	Beschreibung
`template <class ForwardIterator, class T> void fill( ForwardIterator first, ForwardIterator last, const T& value);`	Jedes Element im Bereich `[first,last)` erhält den Wert `value`.
`template <class OutputIterator, class Size, class T> OutputIterator fill_n( OutputIterator first, Size n, const T& value);`	Jedes Element im Bereich `[first,first+n)` erhält den Wert `value`.
`template <class ForwardIterator, class Generator> void generate( ForwardIterator first, ForwardIterator last, Generator gen);`	Jedes Element im Bereich `[first,last)` ruft das Funktionsobjekt `gen` auf und bekommt den Rückgabewert zugewiesen.
`template <class OutputIterator, class Size, class Generator> OutputIterator generate_n( OutputIterator first, Size n, Generator gen);`	Jedes Element im Bereich `[first,first+n)` ruft das Funktionsobjekt `gen` auf und bekommt den Rückgabewert zugewiesen.

Tabelle 5.54 Bereiche füllen

Auch hierzu wieder ein Beispiel, das die Funktionen `fill_n()`, `generate()` und `generate_n()` in der Praxis demonstriert:

```
// stl_algo10.cpp
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
#include <iterator>   // ostream_iterator
#include <cstdlib>    // rand
using namespace std;

int myfunc( void );

int main( ) {
   vector<int> Vector1;
   list<int> List1;
   vector<int>::iterator Viter;
   list<int>::iterator Liter;

   // Gibt 10 × den Wert 666 auf cout aus
   fill_n( ostream_iterator<int>(cout, " "), 10, 666);
   cout << endl;

   // Vector1 mit 10 × dem Wert 0 befüllen
   fill_n( back_inserter(Vector1), 10, 0 );
   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   // Jedes Element, bis auf die letzten 4,
   // mit dem Wert 1 belegen
   fill_n( Vector1.begin(), Vector1.size()-4, 1);
   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   // 10 Zufallszahlen in List1
   generate_n( back_inserter(List1), 10, rand);
   cout << "List1 = ( " ;
   for ( Liter = List1.begin( ) ;
```

```
         Liter != List1.end( ) ; Liter++ )
      cout << *Liter << " ";
   cout << ")" << endl;

   // Mit 10 neuen Werten einer eigenen
   // Funktion überschreiben
   generate( List1.begin(), List1.end(), myfunc);
   cout << "List1 = ( " ;
   for ( Liter = List1.begin( ) ;
         Liter != List1.end( ) ; Liter++ )
      cout << *Liter << " ";
   cout << ")" << endl;
   return 0;
}

int myfunc( void ) {
   static int i = 0;
   return i++;
}
```

Das Programm bei der Ausführung:

```
666 666 666 666 666 666 666 666 666 666
Vector1 = ( 0 0 0 0 0 0 0 0 0 0 )
Vector1 = ( 1 1 1 1 1 1 0 0 0 0 )
List1 = ( 41 1846 6334 2650 1919 1572 1147 2935 26962 2464 )
List1 = ( 0 1 2 3 4 5 6 7 8 9 )
```

Algorithmus	Beschreibung
`template <class ForwardIterator, class T> ForwardIterator remove( ForwardIterator first, ForwardIterator last, const T& value);`	Löscht alle Elemente aus dem Bereich `[first,last)` mit dem Wert `value`. Zurückgegeben wird die Position hinter dem letzten Element im neuen Bereich.
`template <class ForwardIterator, class Predicate> ForwardIterator remove_if( ForwardIterator first, ForwardIterator last, Predicate pred);`	Löscht alle Elemente aus dem Bereich `[first,last)`, auf die die Bedingung `pred` zutrifft. Zurückgegeben wird die Position hinter dem letzten Element im neuen Bereich.

Tabelle 5.55 Löschen von Elementen und Varianten

Algorithmus	Beschreibung
`template <class InputIterator, class OutputIterator, class T> OutputIterator remove_copy( InputIterator first, InputIterator last, OutputIterator result, const T& value);`	Kopiert alle Elemente, die nicht den Wert von `value` besitzen, aus dem Bereich `[first,last)` in die Position `result`. Zurückgegeben wird die Position hinter dem letzten Element im neuen Bereich.
`template <class InputIterator, class OutputIterator, class Predicate> OutputIterator remove_copy_if( InputIterator first, InputIterator last, OutputIterator result, Predicate pred);`	Kopiert alle Elemente, deren Bedingung `pred` nicht `true` ist, aus dem Bereich`[first, last)` in die Position `result`. Zurückgegeben wird die Position hinter dem letzten Element im neuen Bereich.

Tabelle 5.55 Löschen von Elementen und Varianten (Forts.)

Hierzu ein Beispiel, das die Funktionen `remove()`, `remove_copy()` und `remove_copy_if()` in der Praxis demonstriert:

```
// stl_algo11.cpp
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
#include <iterator>   // ostream_iterator
using namespace std;

int main( ) {
   vector<int> Vector1;
   list<int> List1;
   vector<int>::iterator Viter, Viter2;
   list<int>::iterator Liter;

   for(int i = 0; i < 10; i++ )
      Vector1.push_back( i );
   for(int i = 0; i < 10; i++ )
      List1.push_back( i );

   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;
```

```
// Element mit dem Wert 8 entfernen
Viter2 = remove( Vector1.begin(), Vector1.end(), 8 );
// Bis jetzt hat sich noch nichts verändert
cout << "Vector1 = ( " ;
for ( Viter = Vector1.begin( ) ;
      Viter != Vector1.end( ) ; Viter++ )
   cout << *Viter << " ";
cout << ")" << endl;

// Jetzt die Elemente, welche als "removed"
// markiert sind, löschen
Vector1.erase( Viter2, Vector1.end() );
cout << "Vector1 = ( " ;
for ( Viter = Vector1.begin( ) ;
      Viter != Vector1.end( ) ; Viter++ )
   cout << *Viter << " ";
cout << ")" << endl;

// Der ganze Vorgang in einem Rutsch - alle Elemente,
// die kleiner als 5 sind, löschen
Vector1.erase( remove_if( Vector1.begin(), Vector1.end(),
                          bind2nd(less<int>(),5)),
                          Vector1.end());
cout << "Vector1 = ( " ;
for ( Viter = Vector1.begin( ) ;
      Viter != Vector1.end( ) ; Viter++ )
   cout << *Viter << " ";
cout << ")" << endl;

cout << "List1 = ( " ;
for ( Liter = List1.begin( ) ;
      Liter != List1.end( ) ; Liter++ )
   cout << *Liter << " ";
cout << ")" << endl;

// Alle Elemente ausgeben, die kleiner als 5 sind
remove_copy_if(List1.begin(), List1.end(),
               ostream_iterator<int>(cout, " "),
               bind2nd(less<int>(),5));
cout << endl;

list<int> List2;
// Kopiert alle Elemente, die kleiner als 3 sind, nach List2
// Achtung, nicht durch greater täuschen lassen: Wir
// verwenden remove_copy_if und nicht copy_if !!!
remove_copy_if(List1.begin(), List1.end(),
```

```
                  inserter(List2, List2.end()),
                  bind2nd(greater<int>(),3));
   cout << "List2 = ( " ;
   for ( Liter = List2.begin( ) ;
         Liter != List2.end( ) ; Liter++ )
      cout << *Liter << " ";
   cout << ")" << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector1 = ( 0 1 2 3 4 5 6 7 8 9 )
Vector1 = ( 0 1 2 3 4 5 6 7 9 9 )
Vector1 = ( 0 1 2 3 4 5 6 7 9 )
Vector1 = ( 5 6 7 9 )
List1 = ( 0 1 2 3 4 5 6 7 8 9 )
5 6 7 8 9
List2 = ( 0 1 2 3 )
```

Algorithmus	Bedeutung

`template <class ForwardIterator>`
`ForwardIterator unique(`
`  ForwardIterator first,`
`  ForwardIterator last);`
	Entfernt alle gleichen Nachfolger aus dem Bereich `[first,last)`. Zurückgegeben wird die Position hinter dem letzten Element im neuen Bereich.

`template <class ForwardIterator,`
`  class BinaryPredicate>`
`ForwardIterator unique(`
`  ForwardIterator first,`
`  ForwardIterator last,`
`  BinaryPredicate binary_pred);`
	Entfernt alle direkten Nachfolger aus dem Bereich `[first,last)`, wenn die Bedingung `pred` zutrifft. Zurückgegeben wird die Position hinter dem letzten Element im neuen Bereich.

`template <class InputIterator,`
`  class OutputIterator>`
`OutputIterator unique_copy(`
`  InputIterator first,`
`  InputIterator last,`
`  OutputIterator result);`
	Kopiert alle Elemente, die einen gleichen Nachfolger haben, aus dem Bereich `[first,last)` in den Bereich ab `result`. Zurückgegeben wird die Position hinter dem letzten Element im neuen Bereich.

`template <class InputIterator,`
`  class OutputIterator,`
`  class BinaryPredicate>`
`OutputIterator unique_copy(`
`  InputIterator first,`
`  InputIterator last,`
`  OutputIterator result,`
`  BinaryPredicate binary_pred);`
	Kopiert alle Elemente, die einen gleichen Nachfolger haben und auf die die Bedingung `binary_pred` zutrifft, aus dem Bereich `[first,last)` in den Bereich ab `result`. Zurückgegeben wird die Position hinter dem letzten Element im neuen Bereich.

Tabelle 5.56 Entfernen bzw. Kopieren von gleichen aufeinanderfolgenden Elementen einer (Unter-)Sequenz

Hierzu ein Beispiel, das die beiden Funktionen `unique()` und `unique_copy()` in der Praxis zeigen soll:

```
// stl_algo12.cpp
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
#include <iterator>   // ostream_iterator
using namespace std;

int main( ) {
   vector<int> Vector1;
   vector<int>::iterator Viter;
   // Quelldaten
   int data[]={1,2,3,4,5,5,6,7,8,8,3,4,4,5,6,7,7,5,3,2,1,1};
   int dataSize = sizeof(data)/sizeof(int);
   // in Vector kopieren
   copy( data, data+dataSize, back_inserter(Vector1));

   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   // Alle Elemente mit gleichem Vorgänger löschen
   Viter = unique(Vector1.begin(), Vector1.end() );
   // Alle nicht gelöschten Elemente ausgeben
   copy( Vector1.begin(), Viter,
         ostream_iterator<int>(cout, " "));
   cout << endl;

   // Oder gleich auf das Original ...
   Vector1.erase(unique(Vector1.begin(), Vector1.end() ));
   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   // Alle Elemente entfernen, deren Vorgänger
   // größer als das aktuelle Element ist
   Vector1.erase( unique(Vector1.begin(), Vector1.end(),
                         greater<int>()), Vector1.end());
   cout << "Vector1 = ( " ;
```

```
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   // neu initialisieren
   Vector1.clear();
   copy( data, data+dataSize, back_inserter(Vector1));
   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;
   // Elemente ohne Duplikate auf cout ausgeben
   unique_copy( Vector1.begin(), Vector1.end(),
                ostream_iterator<int>(cout, " "));
   cout << endl;

   // Alle Elemente, deren Vorgänger
   // größer ist, auf cout ausgeben

   unique_copy(Vector1.begin(), Vector1.end(),
               ostream_iterator<int>(cout, " "),
               greater<int>() );
   cout << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector1 = ( 1 2 3 4 5 5 6 7 8 8 3 4 4 5 6 7 7 5 3 2 1 1 )
1 2 3 4 5 6 7 8 3 4 5 6 7 5 3 2 1
Vector1 = ( 1 2 3 4 5 6 7 8 3 4 5 6 7 5 3 2 1 5 3 2 1 )
Vector1 = ( 1 2 3 4 5 6 7 8 )
Vector1 = ( 1 2 3 4 5 5 6 7 8 8 3 4 4 5 6 7 7 5 3 2 1 1 )
1 2 3 4 5 6 7 8 3 4 5 6 7 5 3 2 1
1 2 3 4 5 5 6 7 8 8
```

Algorithmus	Beschreibung

`template <class ForwardIterator>`
`void` **`rotate`**`(`
`  ForwardIterator first,`
`  ForwardIterator middle,`
`  ForwardIterator last);`
	Rotiert die Elemente aus dem Bereich `[first,last)` um `middle-first` Positionen nach links. Auf jeden Fall wird dem Element `*first` das Element `*middle` zugewiesen. `middle` muss hier im Bereich `[first,last)` sein. Vorn herausfallende Elemente werden hinten wieder eingefügt.

Tabelle 5.57 Ring-Shift-Algorithmen

Algorithmus	Beschreibung
`template <class ForwardIterator, class OutputIterator> OutputIterator rotate_copy( ForwardIterator first, ForwardIterator middle, ForwardIterator last, OutputIterator result);`	Kopiert die Elemente aus dem Bereich `[first,last)` an den Anfang des Bereichs `result` und rotiert dabei die Elemente um `middle-first` Positionen nach links. Auf jeden Fall wird dem Element `*first` das Element `*middle` zugewiesen. Die Bereiche `[first, last)` und `result` dürfen sich nicht überlappen. Zurückgegeben wird `result+(last-first)`.

Tabelle 5.57 Ring-Shift-Algorithmen (Forts.)

Hierzu ein Beispiel, das die beiden Algorithmen `rotate()` und `rotate_copy()` in der Praxis demonstriert:

```
// stl_algo13.cpp
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
#include <iterator>   // ostream_iterator
using namespace std;
int main( ) {
   vector<int> Vector1;
   vector<int>::iterator Viter;

   for(int i = 0; i < 10; i++ )
      Vector1.push_back( i );

   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   // Um ein Element nach links verschieben
   rotate( Vector1.begin(), Vector1.begin()+1,
           Vector1.end());
   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   // Um 4 Elemente nach rechts verschieben
```

```
   rotate( Vector1.begin(), Vector1.end()-4,
           Vector1.end());

   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   // Um 4 Elemente nach links verschieben und ausgeben
   Viter=Vector1.begin();
   advance( Viter, 4 );
   cout << "Vector1 = ( " ;
   rotate_copy(Vector1.begin(), Viter,
               Vector1.end(),
               ostream_iterator<int>(cout," "));
   cout << ")" << endl;

   // Um ein Element nach rechts verschieben und ausgeben
   Viter=Vector1.end();
   advance( Viter, -1 );
   cout << "Vector1 = ( " ;
   rotate_copy(Vector1.begin(), Viter,
               Vector1.end(),
               ostream_iterator<int>(cout," "));
   cout << ")" << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector1 = ( 0 1 2 3 4 5 6 7 8 9 )
Vector1 = ( 1 2 3 4 5 6 7 8 9 0 )
Vector1 = ( 7 8 9 0 1 2 3 4 5 6 )
Vector1 = ( 1 2 3 4 5 6 7 8 9 0 )
Vector1 = ( 6 7 8 9 0 1 2 3 4 5 )
```

Algorithmus	Beschreibung

`template <`
`  class RandomAccessIterator>`
`void` **`random_shuffle`**`(`
`  RandomAccessIterator first,`
`  RandomAccessIterator last);`
	Damit mischen Sie die Elemente aus dem Bereich `[first,last)` einer Sequenz bzw. einer zufälligen Reihenfolge. Der Algorithmus verwendet dabei eine nicht vom Standard vorgeschriebene Zufallsfunktion.

Tabelle 5.58 Elemente mischen (verteilen)

Algorithmus	Beschreibung

`template <`
`  class RandomAccessIterator,`
`  class RandomNumberGenerator>`
`void random_shuffle(`
`  RandomAccessIterator first,`
`  RandomAccessIterator last,`
`  RandomNumberGenerator& rand);`
	Dito, nur können hier die Elemente mit Hilfe eines Zufallsgenerators `rand` verteilt werden.

Tabelle 5.58 Elemente mischen (verteilen) (Forts.)

Hier ein Beispiel zur Funktion `random_shuffle()` in der Praxis:

```
// stl_algo14.cpp
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
#include <iterator>   // ostream_iterator
using namespace std;

int main( ) {
   vector<int> Vector1;
   vector<int>::iterator Viter;

   for(int i = 0; i < 10; i++ )
      Vector1.push_back( i );
   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   // Durchmischen
   random_shuffle( Vector1.begin(), Vector1.end() );
   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   // Wieder sortieren
   sort( Vector1.begin(), Vector1.end() );
   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
```

```
      cout << *Viter << " ";
   cout << ")" << endl;

   // Nochmals durchmischen
   random_shuffle( Vector1.begin(), Vector1.end() );
   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector1 = ( 0 1 2 3 4 5 6 7 8 9 )
Vector1 = ( 8 1 9 2 0 5 7 3 4 6 )
Vector1 = ( 0 1 2 3 4 5 6 7 8 9 )
Vector1 = ( 6 4 9 7 3 0 1 8 5 2 )
```

Algorithmus	Beschreibung

`template <`
`  class BidirectionalIterator,`
`  class Predicate>`
`BidirectionalIterator` **`partition`**`(`
`  BidirectionalIterator first,`
`  BidirectionalIterator last,`
`  Predicate pred);`
	Zerlegt den Bereich `[first,last)` in zwei Bereiche, so dass alle Elemente dem Kriterium `pred` genügen. Gibt einen Iterator zurück, der auf den Anfang des zweiten Bereichs zeigt. Typischer Anwendungsfall: Quicksort-Algorithmus

`template <`
`  class BidirectionalIterator,`
`  class Predicate>`
`BidirectionalIterator`
`stable_partition``(`
`  BidirectionalIterator first,`
`  BidirectionalIterator last,`
`  Predicate pred);`
	Dito, nur garantiert diese Version darüber hinaus, dass die relative Ordnung der Elemente innerhalb des Bereichs `[first,last)` erhalten bleibt.

`template <`
`  class RandomAccessIterator>`
`void` **`nth_element`**`(`
`  RandomAccessIterator first,`
`  RandomAccessIterator nth,`
`  RandomAccessIterator last);`
	Zerlegt die Elemente im Bereich `[first,last)` anhand des Elements an der Position `nth` in einen linken und einen rechten Teil. Auf die linke Seite kommen die Elemente, die kleiner als `nth` sind, und auf die rechte Seite werden alle Elemente kopiert, die größer sind als das Vergleichselement `nth`.

Tabelle 5.59 Bereich in zwei Teile zerlegen

Algorithmus	Beschreibung

`template <`
`  class RandomAccessIterator,`
`  class Compare>`
`void nth_element(`
`  RandomAccessIterator first,`
`  RandomAccessIterator nth,`
`  RandomAccessIterator last,`
`  Compare comp);`
	Zerlegt die Elemente im Bereich `[first,last)` anhand des Elements an der Position `nth` und des Funktionsobjekts `comp` in einen linken und einen rechten Teil. Auf die linke Seite kommen die Elemente, die kleiner als `nth` sind und dem Vergleichskriterium `comp` entsprechen, und auf die rechte Seite werden alle Elemente kopiert, die größer sind als das Vergleichselement `nth` und dem Vergleichskriterium `comp` entsprechen.

Tabelle 5.59 Bereich in zwei Teile zerlegen (Forts.)

Hier ein Beispiel zu den Algorithmen `partition()` und `stable_partition()` in der Praxis:

```
// stl_algo15.cpp
#include <vector>
#include <algorithm>
#include <iostream>
#include <iterator>   // ostream_iterator
using namespace std;

int main( ) {
   vector<int> Vector1, Vector2;
   vector<int>::iterator Viter;

   for(int i = 0; i < 10; i++ )
      Vector1.push_back( i );
   for(int i = 0; i < 10; i++ )
      Vector2.push_back( i );

   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   cout << "Vector2 = ( " ;
   for ( Viter = Vector2.begin( ) ;
         Viter != Vector2.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   // Alle Elemente, die durch 2 teilbar sind, auf
   // die linke Seite
   Viter = partition( Vector1.begin(), Vector1.end(),
```

```
                    not1(bind2nd(modulus<int>(),2)));
   cout << "Die Mitte: " << *Viter << endl;
   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   // Alle Elemente, die durch 2 teilbar sind, auf die
   // linke Seite. Im Gegensatz zu partition sind die
   // Elemente zusätzlich noch sortiert ...
   Viter = stable_partition( Vector2.begin(), Vector2.end(),
                    not1(bind2nd(modulus<int>(),2)));
   cout << "Die Mitte: " << *Viter << endl;
   cout << "Vector2 = ( " ;
   for ( Viter = Vector2.begin( ) ;
         Viter != Vector2.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector1 = ( 0 1 2 3 4 5 6 7 8 9 )
Vector2 = ( 0 1 2 3 4 5 6 7 8 9 )
Die Mitte: 5
Vector1 = ( 0 8 2 6 4 5 3 7 1 9 )
Die Mitte: 1
Vector2 = ( 0 2 4 6 8 1 3 5 7 9 )
```

Algorithmen zum Sortieren

Alle Algorithmen zum Sortieren sind in zwei Versionen vorhanden. Eine Version vergleicht die Elemente mit dem <-Operator (und muss gegebenenfalls implementiert werden), und die andere Version verwendet ein Funktionsobjekt (`comp`) zum Vergleichen der Elemente. Natürlich können Sie statt eines Funktionsobjekts auch eine eigene Funktion verwenden.

Algorithmus	Beschreibung

`template <`
`  class RandomAccessIterator>`
`void sort(`
`  RandomAccessIterator first,`
`  RandomAccessIterator last);`
	Sortiert die Elemente im Bereich `[first,last)` anhand des <-Operators.

Tabelle 5.60 Algorithmen für das Sortieren

Algorithmus	Beschreibung

`template <`
`  class RandomAccessIterator,`
`  class Compare>`
`void sort(`
`  RandomAccessIterator first,`
`  RandomAccessIterator last,`
`  Compare comp);`
	Sortiert die Elemente im Bereich `[first,last)` anhand des Funktionsobjekts `comp`.

`template <`
`  class RandomAccessIterator>`
`void stable_sort(`
`  RandomAccessIterator first,`
`  RandomAccessIterator last);`
	Dasselbe wie bei `sort()`, nur ist dieser Algorithmus besser im Laufzeitverhalten im schlechtesten Fall (*worst case*) und daher meist `sort()` vorzuziehen.

`template <`
`  class RandomAccessIterator,`
`  class Compare>`
`void stable_sort(`
`  RandomAccessIterator first,`
`  RandomAccessIterator last,`
`  Compare comp);`
	Dasselbe wie bei `sort()`, nur ist dieser Algorithmus besser im Laufzeitverhalten im schlechtesten Fall (*worst case*) und daher meist `sort()` vorzuziehen.

`template <`
`  class RandomAccessIterator>`
`void partial_sort(`
`  RandomAccessIterator first,`
`  RandomAccessIterator middle,`
`  RandomAccessIterator last);`
	Damit sortieren Sie die ersten (`middle-first`) Elemente im Bereich `[first,last)` anhand des `<`-Operators. Der Rest bleibt unsortiert.

`template <`
`  class RandomAccessIterator,`
`  class Compare>`
`void partial_sort(`
`  RandomAccessIterator first,`
`  RandomAccessIterator middle,`
`  RandomAccessIterator last,`
`  Compare comp);`
	Damit sortieren Sie die ersten (`middle-first`) Elemente im Bereich `[first,last)` anhand des Funktionsobjekts `comp`. Der Rest bleibt unsortiert.

`template <`
`  class InputIterator,`
`  class RandomAccessIterator>`
`RandomAccessIterator`
`partial_sort_copy(`
`  InputIterator first,`
`  InputIterator last,`
`  RandomAccessIterator`
`  result_first,`
`  RandomAccessIterator`
`  result_last);`
	Sortiert die Elemente aus dem Bereich `[first,last)` anhand des `<`-Operators und kopiert das Ergebnis in den Bereich `[res_first,res_last)`. Es werden nur so viele Elemente kopiert, wie die kleinere Zahl von (`last-first`) oder (`res_last-res_first`) festlegt. Zurückgegeben wird der kleinere der beiden Iteratoren.

Tabelle 5.60 Algorithmen für das Sortieren (Forts.)

Algorithmus	Beschreibung

`template <`
`  class InputIterator,`
`  class RandomAccessIterator,`
`  class StrictWeakOrdering>`
`RandomAccessIterator`
`partial_sort_copy(`
`  InputIterator first,`
`  InputIterator last,`
`  RandomAccessIterator`
`  result_first,`
`  RandomAccessIterator`
`  result_last,`
`  Compare comp);`
	Sortiert die Elemente aus dem Bereich `[first,last)` anhand des Funktionsobjekts `comp` und kopiert das Ergebnis in den Bereich `[res_first,res_last)`. Es werden nur so viele Elemente kopiert, wie die kleinere Zahl von (`last-first`) oder (`res_last-res_first`) festlegt. Zurückgegeben wird der kleinere der beiden Iteratoren.

`template <class ForwardIterator>`
`bool is_sorted(`
`  ForwardIterator first,`
`  ForwardIterator last);`
	Dieser Algorithmus gibt `true` zurück, wenn der Bereich `[first,last)` bereits in aufsteigender Reihenfolge sortiert ist, ansonsten `false`.

`template <`
`  class ForwardIterator,`
`  class StrictWeakOrdering>`
`bool is_sorted(`
`  ForwardIterator first,`
`  ForwardIterator last,`
`  StrictWeakOrdering comp);`
	Dieser Algorithmus gibt `true` zurück, wenn der Bereich `[first,last)` anhand des Funktionsobjekts `comp` sortiert ist, ansonsten `false`.

Tabelle 5.60 Algorithmen für das Sortieren (Forts.)

[»] Hinweis

Beachten Sie bitte, dass die Algorithmen, die Random-Access-Iteratoren verwenden, nur für Container geeignet sind, die einen wahlfreien Zugriff auf die Elemente erlauben. Dies sind Container wie `vector` oder `deque`, aber nicht `list`. Die Container-Klasse `list` bietet daher eine eigene Methode `list::sort()` zum Sortieren an.

Hierzu ein Listing, das die Algorithmen `sort()`, `partial_sort()` und `partial_sort_copy()` in der Praxis demonstrieren soll:

```
// stl_algo16.cpp
#include <vector>
#include <algorithm>
#include <iostream>
#include <iterator>   // ostream_iterator
using namespace std;

int main( ) {
   vector<int> Vector1, Vector2, Vector3, Vector4(10);
```

```
vector<int>::iterator Viter;
// Quelldaten
int data[]={5,9,3,4,5,5,6,7,8,8,3,4,4,5,6,7,7,5,3,2,1,1};
int dataSize = sizeof(data)/sizeof(int);
// in Vector kopieren
copy( data, data+dataSize, back_inserter(Vector1));
copy( data, data+dataSize, back_inserter(Vector2));
copy( data, data+dataSize, back_inserter(Vector3));

cout << "Vector1 = ( " ;
for ( Viter = Vector1.begin( ) ;
      Viter != Vector1.end( ) ; Viter++ )
   cout << *Viter << " ";
cout << ")" << endl;

cout << "Vector2 = ( " ;
for ( Viter = Vector2.begin( ) ;
      Viter != Vector2.end( ) ; Viter++ )
   cout << *Viter << " ";
cout << ")" << endl;

cout << "Vector3 = ( " ;
for ( Viter = Vector3.begin( ) ;
      Viter != Vector3.end( ) ; Viter++ )
   cout << *Viter << " ";
cout << ")" << endl;

sort( Vector1.begin(), Vector1.end());

cout << "Vector1 = ( " ;
for ( Viter = Vector1.begin( ) ;
      Viter != Vector1.end( ) ; Viter++ )
   cout << *Viter << " ";
cout << ")" << endl;

// Die ersten 8 Elemente sortieren
partial_sort(Vector2.begin(), Vector2.begin()+8,
             Vector2.end());

cout << "Vector2 = ( " ;
for ( Viter = Vector2.begin( ) ;
      Viter != Vector2.end( ) ; Viter++ )
   cout << *Viter << " ";
cout << ")" << endl;
```

```
   // Die letzten 8 Elemente absteigend sortieren
   partial_sort( Vector3.begin(), Vector3.end()-8,
                 Vector3.end(), less<int>());

   cout << "Vector3 = ( " ;
   for ( Viter = Vector3.begin( ) ;
         Viter != Vector3.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   // Kopiert Elemente von data sortiert nach Vector4
   partial_sort_copy( data, data+dataSize,
                      Vector4.begin(), Vector4.end() );

   cout << "Vector4 = ( " ;
   for ( Viter = Vector4.begin( ) ;
         Viter != Vector4.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector1 = ( 5 9 3 4 5 5 6 7 8 8 3 4 4 5 6 7 7 5 3 2 1 1 )
Vector2 = ( 5 9 3 4 5 5 6 7 8 8 3 4 4 5 6 7 7 5 3 2 1 1 )
Vector3 = ( 5 9 3 4 5 5 6 7 8 8 3 4 4 5 6 7 7 5 3 2 1 1 )
Vector1 = ( 1 1 2 3 3 3 4 4 4 5 5 5 5 5 6 6 7 7 7 8 8 9 )
Vector2 = ( 1 1 2 3 3 3 4 4 9 8 8 7 6 5 6 7 7 5 5 5 5 4 )
Vector3 = ( 1 1 2 3 3 3 4 4 4 5 5 5 5 5 9 8 8 7 7 7 6 6 )
Vector4 = ( 1 1 2 3 3 3 4 4 4 5 )
```

Algorithmen für die binäre Suche

Voraussetzung für diese Algorithmen ist, dass die Elemente in einer sortierten Reihenfolge vorhanden sind und der Zugriff über einen Random-Access-Iterator erfolgt.

Algorithmus	Beschreibung

`template <class ForwardIterator, class T>`
`bool binary_search(`
`  ForwardIterator first,`
`  ForwardIterator last,`
`  const T& value);`
	Sucht nach dem Wert `value` im Bereich `[first,last)` mit dem binären Suchverfahren. Gibt `true` zurück, falls `value` gefunden wurde, ansonsten `false`.

Tabelle 5.61 Binäre Suche

Algorithmus	Beschreibung
`template <class ForwardIterator, class T, class Compare> bool binary_search( ForwardIterator first, ForwardIterator last, const T& value, Compare comp);`	Dito, nur werden die Elemente zusätzlich noch mit dem Funktionsobjekt `comp` verglichen.
`template <class ForwardIterator, class T> ForwardIterator lower_bound( ForwardIterator first, ForwardIterator last, const T& value);`	Liefert aus dem Bereich `[first,last)` die Position des Elements zurück, das größer oder gleich dem Wert `value` ist.
`template <class ForwardIterator, class T, class Compare> ForwardIterator lower_bound( ForwardIterator first, ForwardIterator last, const T& value, Compare comp);`	Liefert aus dem Bereich `[first,last)` die Position des Elements zurück, das größer oder gleich dem Wert `value` ist und zusätzlich dem Kriterium von `comp` entspricht.
`template <class ForwardIterator, class T> ForwardIterator upper_bound( ForwardIterator first, ForwardIterator last, const T& value);`	Liefert aus dem Bereich `[first,last)` die Position des Elements zurück, das kleiner als `value` ist.
`template <class ForwardIterator, class T, class Compare> ForwardIterator upper_bound( ForwardIterator first, ForwardIterator last, const T& value, Compare comp);`	Liefert aus dem Bereich `[first,last)` die Position des Elements zurück, das kleiner als `value` ist und zusätzlich dem Kriterium von `comp` entspricht.
`template <class ForwardIterator, class T> pair<ForwardIterator, ForwardIterator> equal_range( ForwardIterator first, ForwardIterator last, const T& value);`	Gibt die Iteratoren `lower_bound(first, last,val)` und `upper_bound(first, last,val)` als Paar zurück.

Tabelle 5.61 Binäre Suche (Forts.)

Algorithmus	Beschreibung
`template <class ForwardIterator, class T, class Compare> pair<ForwardIterator, ForwardIterator> equal_range( ForwardIterator first, ForwardIterator last, const T& value, Compare comp);`	Dito, aber zusätzlich werden die Elemente mit dem Funktionsobjekt `comp` verglichen.

Tabelle 5.61 Binäre Suche (Forts.)

Hierzu ein Listing, das die Algorithmen `binary_search()` und `lower_bound()` demonstrieren soll:

```
// stl_algo17.cpp
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main( ) {
   vector<int> Vector1, Vector2;
   vector<int>::iterator Viter;
   // Quelldaten
   int data[]={5,9,3,4,5,5,6,7,8,8,3,4,4,5,6,7,7,5,3,2,1,1};
   int dataSize = sizeof(data)/sizeof(int);
   // in Vector kopieren
   copy( data, data+dataSize, back_inserter(Vector1));
   // Elemente sortieren
   sort(Vector1.begin(), Vector1.end());

   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   // Existiert der Wert 8?
   if( binary_search(Vector1.begin(), Vector1.end(), 8 ) ) {
      cout << "Ein Element mit dem Wert 8 wurde gefunden"
           << endl;
   }
   else {
```

```
      cout << "Es gibt kein Element mit dem Wert 8" << endl;
   }
   // Existiert der Wert 8 unter den ersten 10 Elementen?
   if( binary_search(Vector1.begin(),
                     Vector1.begin()+10, 8 ) ) {
      cout << "Ein Element mit dem Wert 8 wurde gefunden"
           << endl;
   }
   else {
      cout << "Es gibt kein Element mit dem Wert 8"
           << "unter den ersten 10" << endl;
   }
   // Liefert Pos. zurück, deren Wert größer gleich 5 ist
   Viter = lower_bound(Vector1.begin(),Vector1.end(), 5);
   // ab dieser Position alles nach Vector2 kopieren
   copy( Viter, Vector1.end(), back_inserter(Vector2));
   cout << "Vector2 = ( " ;
   for ( Viter = Vector2.begin( ) ;
         Viter != Vector2.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector1 = ( 1 1 2 3 3 3 4 4 4 5 5 5 5 5 6 6 7 7 7 8 8 9 )
Ein Element mit dem Wert 8 wurde gefunden
Es gibt kein Element mit dem Wert 8 unter den ersten 10
Vector2 = ( 5 5 5 5 5 6 6 7 7 7 8 8 9 )
```

Algorithmen zum Verschmelzen (Mischen)

Beim Verschmelzen (oder auch Mischen) werden zwei sortierte Sequenzen zu einer vermischt. Dabei werden die einzelnen Elemente beider Sequenzen Element für Element verglichen und je nach Sortierkriterium (standardmäßig: *Kleiner als*) in die Ausgabesequenz sortiert übergeben. »Element für Element« bedeutet, dass die jeweils ersten, zweiten usw. Elemente bis zu den letzten Elementen miteinander verglichen werden.

Algorithmus	Beschreibung
`template <class InputIterator1, class InputIterator2, class OutputIterator> OutputIterator merge( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result );`	Mischt zwei sortierte Bereiche `[first1, last1)` und `[first2,last2)` in aufsteigender Form in den Bereich `[result, result+(last1-first1)+(last2-first2))`. Quell- und Zielbereich dürfen sich dabei nicht überlappen. Zurückgegeben wird die Position hinter dem letzten Element im Zielbereich.
`template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare> OutputIterator merge( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp );`	Dito, nur können Sie hier mit `comp` ein Funktionsobjekt verwenden, mit dem Sie das Misch- bzw. Sortierkriterium angeben.
`template < class BidirectionalIterator> void inplace_merge( BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last);`	Damit mischen Sie die beiden (sortierten) Bereiche `[first,middle)` und `[middle, last)` in aufsteigender Reihenfolge in den Bereich `[first,last)`.
`template < class BidirectionalIterator, class Compare> void inplace_merge( BidirectionalIterator first, BidirectionalIterator middle, BidirectionalIterator last, Compare comp );`	Dito, nur können Sie hier mit `comp` ein Funktionsobjekt verwenden, mit dem Sie das Misch- bzw. Sortierkriterium angeben.

Tabelle 5.62 Verschmelzen (Mischen) von Elementen

Hierzu ein Beispiel, das die Funktion `merge()` in der Praxis zeigen soll:

```
// stl_algo18.cpp
#include <vector>
#include <algorithm>
#include <iostream>
#include <iterator>
using namespace std;
```

```
int main() {
   vector<int> Vector1, Vector2, Vector3;
   vector<int>::iterator Viter;
   // Quelldaten
   int data[]={5,9,3,4,7,5,3,2,1,6};
   int dataSize = sizeof(data)/sizeof(int);
   // in Vector kopieren
   copy( data, data+dataSize, back_inserter(Vector1));
   for(int i = 0; i < 10; i++ )
      Vector2.push_back( i );

   // Elemente sortieren
   sort(Vector1.begin(), Vector1.end());

   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;
   cout << "Vector2 = ( " ;
   for ( Viter = Vector2.begin( ) ;
         Viter != Vector2.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   cout << "Gemischt: ";
   merge( Vector1.begin(), Vector1.end(),
          Vector2.begin(), Vector2.end(),
          ostream_iterator<int>(cout, " "));
   cout << endl;
   // ... oder gleich in Vector3 kopieren
   merge( Vector1.begin(), Vector1.end(),
          Vector2.begin(), Vector2.end(),
          back_inserter(Vector3));

   cout << "Vector3 = ( " ;
   for ( Viter = Vector3.begin( ) ;
         Viter != Vector3.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector1 = ( 1 2 3 3 4 5 5 6 7 9 )
Vector2 = ( 0 1 2 3 4 5 6 7 8 9 )
Gemischt: 0 1 1 2 2 3 3 3 4 4 5 5 5 6 6 7 7 8 9 9
Vector3 = ( 0 1 1 2 2 3 3 3 4 4 5 5 5 6 6 7 7 8 9 9 )
```

Mengenoperationen auf sortierten Strukturen

Algorithmen	Beschreibung

`template <class InputIterator1,`
`class InputIterator2>`
`bool includes(`
`InputIterator1 first1,`
`InputIterator1 last1,`
`InputIterator2 first2,`
`InputIterator2 last2);`
	Gibt `true` zurück, wenn die Elemente `[first2,last2)` in `[first1,last1)` enthalten sind, ansonsten `false`. Als Vergleich wird der <-Operator verwendet (Teilmenge).

`template <class InputIterator1,`
`class InputIterator2,`
`class Compare>`
`bool includes(`
`InputIterator1 first1,`
`InputIterator1 last1,`
`InputIterator2 first2,`
`InputIterator2 last2,`
`Compare comp);`
	Dito, nur können Sie hier mit `comp` ein Funktionsobjekt verwenden, mit dem Sie ein anderes Kriterium als den <-Operator angeben können.

`template <class InputIterator1,`
`class InputIterator2,`
`class OutputIterator>`
`OutputIterator set_union(`
`InputIterator1 first1,`
`InputIterator1 last1,`
`InputIterator2 first2,`
`InputIterator2 last2,`
`OutputIterator result);`
	Erstellt eine sortierte Menge aus einem oder beiden Bereichen `[first1,last1)` und `[first2,last2)` und kopiert die Elemente in den Bereich `result` (Vereinigungsmenge).

`template <class InputIterator1,`
`class InputIterator2,`
`class OutputIterator,`
`class Compare>`
`OutputIterator set_union(`
`InputIterator1 first1,`
`InputIterator1 last1,`
`InputIterator2 first2,`
`InputIterator2 last2,`
`OutputIterator result,`
`Compare comp);`
	Dito, nur können Sie hier mit `comp` ein Funktionsobjekt verwenden, mit dem Sie ein anderes Kriterium als den <-Operator angeben können.

Tabelle 5.63 Mengenoperationen auf sortierten Strukturen

Algorithmen	Beschreibung
`template <class InputIterator1, class InputIterator2, class OutputIterator> OutputIterator set_intersection( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);`	Erstellt eine sortierte Menge aus den beiden Bereichen `[first1,last1)` und `[first2, last2)` und kopiert die Elemente in den Bereich `result` (Schnittmenge).
`template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare> OutputIterator set_intersection( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);`	Dito, nur können Sie hier mit `comp` ein Funktionsobjekt verwenden, mit dem Sie ein anderes Kriterium als den `<`-Operator angeben können.
`template <class InputIterator1, class InputIterator2, class OutputIterator> OutputIterator set_difference( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);`	Erstellt eine sortierte Menge aus Elementen, die in dem Bereich `[first1,last1)`, aber nicht in `[first2,last2)` vorkommen, und kopiert diese in den Bereich `result` (Differenzmenge).
`template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare> OutputIterator set_difference( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);`	Dito, nur können Sie hier mit `comp` ein Funktionsobjekt verwenden, mit dem Sie ein anderes Kriterium als den `<`-Operator angeben können.

Tabelle 5.63 Mengenoperationen auf sortierten Strukturen (Forts.)

Algorithmen	Beschreibung
`template <class InputIterator1, class InputIterator2, class OutputIterator> OutputIterator set_symmetric_difference( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);`	Kopiert Elemente aus den Bereichen `[first1,last1)`, die nicht in`[first2, last2)` enthalten sind, und Elemente aus den Bereichen `[first2,last2)`, die wiederum nicht in `[first1,last1)` enthalten sind, in den Bereich, der mit `result` beginnt.
`template <class InputIterator1, class InputIterator2, class OutputIterator, class Compare> OutputIterator set_symmetric_difference( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result, Compare comp);`	Dito, nur können Sie hier mit `comp` ein Funktionsobjekt verwenden, mit dem Sie ein anderes Kriterium als den <-Operator angeben können.

Tabelle 5.63 Mengenoperationen auf sortierten Strukturen (Forts.)

Hierzu ein Beispiel, das alle in Tabelle 5.63 beschriebenen Funktionen in der Praxis demonstriert:

```
// stl_algo19.cpp
#include <vector>
#include <algorithm>
#include <iostream>
#include <iterator>
using namespace std;

int main() {
   vector<int> Vector1, Vector2, Vector3,
               Vector4, Vector5, Vector6;
   vector<int>::iterator Viter;
   // Quelldaten
   int data[]={5,9,3};
   int dataSize = sizeof(data)/sizeof(int);
   // in Vector kopieren
   copy( data, data+dataSize, back_inserter(Vector1));
   for(int i = 0; i < 10; i++ )
      Vector2.push_back( i );
```

```
// Elemente sortieren
sort(Vector1.begin(), Vector1.end());

cout << "Vector1 = ( " ;
for ( Viter = Vector1.begin( ) ;
      Viter != Vector1.end( ) ; Viter++ )
   cout << *Viter << " ";
cout << ")" << endl;
cout << "Vector2 = ( " ;
for ( Viter = Vector2.begin( ) ;
      Viter != Vector2.end( ) ; Viter++ )
   cout << *Viter << " ";
cout << ")" << endl;

// Überprüfen, ob alle Elemente von
// Vector1 auch in Vector2 vorkommen
if( includes( Vector2.begin(), Vector2.end(),
              Vector1.begin(), Vector1.end())){
   cout << "Alle Elemente von Vector1 sind auch"
        << " in Vector2 vorhanden" << endl;
}
else {
   cout << "Nicht alle Elemente von Vector1 sind"
        << " in Vector2 vorhanden" << endl;
}
Vector1.push_back(13);
Vector1.push_back(11);
// Elemente sortieren
sort(Vector1.begin(), Vector1.end());

cout << "Vector1 = ( " ;
for ( Viter = Vector1.begin( ) ;
      Viter != Vector1.end( ) ; Viter++ )
   cout << *Viter << " ";
cout << ")" << endl;

cout << "Vector2 = ( " ;
for ( Viter = Vector2.begin( ) ;
      Viter != Vector2.end( ) ; Viter++ )
   cout << *Viter << " ";
cout << ")" << endl;

set_union( Vector2.begin(), Vector2.end(),
           Vector1.begin(), Vector1.end(),
           back_inserter(Vector3));
```

```
   cout << "Vector3 = ( " ;
   for ( Viter = Vector3.begin( ) ;
         Viter != Vector3.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;


   set_intersection( Vector2.begin(), Vector2.end(),
                     Vector1.begin(), Vector1.end(),
                     back_inserter(Vector4));

   cout << "Vector4 = ( " ;
   for ( Viter = Vector4.begin( ) ;
         Viter != Vector4.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   set_difference  ( Vector2.begin(), Vector2.end(),
                     Vector1.begin(), Vector1.end(),
                     back_inserter(Vector5));

   cout << "Vector5 = ( " ;
   for ( Viter = Vector5.begin( ) ;
         Viter != Vector5.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   set_symmetric_difference( Vector2.begin(), Vector2.end(),
                             Vector1.begin(), Vector1.end(),
                             back_inserter(Vector6));

   cout << "Vector6 = ( " ;
   for ( Viter = Vector6.begin( ) ;
         Viter != Vector6.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector1 = ( 3 5 9 )
Vector2 = ( 0 1 2 3 4 5 6 7 8 9 )
Alle Elemente von Vector1 sind auch in Vector2 vorhanden
Vector1 = ( 3 5 9 11 13 )
```

```
Vector2 = ( 0 1 2 3 4 5 6 7 8 9 )
Vector3 = ( 0 1 2 3 4 5 6 7 8 9 11 13 )
Vector4 = ( 3 5 9 )
Vector5 = ( 0 1 2 4 6 7 8 )
Vector6 = ( 0 1 2 4 6 7 8 11 13 )
```

Heap-Algorithmen

Bei einem Heap handelt es sich um eine Form des binären Baums mit minimaler Höhe, bei dem das größte (bzw. kleinste) Element immer in der Wurzel gespeichert wird. Die Priority-Queue aus Abschnitt 5.3.5, »Container«, basiert beispielsweise auf einem solchen Heap. Gewöhnlich wird ein Heap verwendet, um den Heap-Sort-Algorithmus zu implementieren, ein sehr schnelles und stabiles Sortierverfahren, das selbst im schlechtesten Fall proportional zu `n*log(n)` (`n` = Anzahl der Elemente) ist.

Wie schon bei den anderen Algorithmen zu STL sind auch hier keine genauen Angaben zu den Containern nötig. Es werden lediglich zwei Random-Access-Operatoren übergeben, die den Heap-Bereich markieren, um damit zu arbeiten. Auch hier ist, wie meistens, das Prioritätskriterium `less<T>`. Dieses Kriterium lässt sich auch wieder den Gegebenheiten anpassen und kann ein eigenes Funktionsobjekt verwenden.

Algorithmen	Beschreibung

`template <`
`  class RandomAccessIterator>`
`void pop_heap(`
`  RandomAccessIterator first,`
`  RandomAccessIterator last);`
	Entnimmt das größte Element (hier `*first`) aus dem Bereich `[first,last)` aus dem Heap. `[first,last)` muss ein gültiger Heap-Bereich sein.

`template <`
`  class RandomAccessIterator,`
`  class Compare>`
`void pop_heap(`
`  RandomAccessIterator first,`
`  RandomAccessIterator last,`
`  Compare comp);`
	Dito, nur kann hier mit `comp` ein anderes Prioritätskriterium als `less<T>` als Funktionsobjekt verwendet werden.

`template <`
`  class RandomAccessIterator>`
`void push_heap(`
`  RandomAccessIterator first,`
`  RandomAccessIterator last);`
	Fügt ein neues Element in die Position `last-1` im Heap ein. Voraussetzung dafür ist, dass `[first,last-2)` einen gültigen Heap darstellen.

Tabelle 5.64 Heap-Algorithmen

Algorithmen	Beschreibung

`template <`
`  class RandomAccessIterator,`
`  class Compare>`
`void push_heap(`
`  RandomAccessIterator first,`
`  RandomAccessIterator last,`
`  Compare comp);`
	Dito, nur kann hier mit `comp` ein anderes Prioritätskriterium als `less<T>` als Funktionsobjekt verwendet werden.

`template <`
`  class RandomAccessIterator>`
`void make_heap(`
`  RandomAccessIterator first,`
`  RandomAccessIterator last);`
	Erzeugt aus dem Bereich `[first,last)` einen Heap.

`template <`
`  class RandomAccessIterator,`
`  class Compare>`
`void make_heap(`
`  RandomAccessIterator first,`
`  RandomAccessIterator last,`
`  Compare comp);`
	Dito, nur kann hier mit `comp` ein anderes Prioritätskriterium als `less<T>` als Funktionsobjekt verwendet werden.

`template <`
`  class RandomAccessIterator>`
`void sort_heap(`
`  RandomAccessIterator first,`
`  RandomAccessIterator last);`
	Hiermit verwandeln Sie den Heap mit dem Bereich `[first,last)` in eine sortierte Sequenz. Der Bereich `[first,last)` ist anschließend kein Heap mehr. Die Sequenz wird aufsteigend (`less<T>`) sortiert.

`template <`
`  class RandomAccessIterator,`
`  class Compare>`
`void sort_heap(`
`  RandomAccessIterator first,`
`  RandomAccessIterator last,`
`  Compare comp);`
	Dito, nur kann hier mit `comp` ein anderes Prioritätskriterium als `less<T>` als Funktionsobjekt verwendet werden.

Tabelle 5.64 Heap-Algorithmen (Forts.)

[»] **Hinweis**

Es ist zwar kein Standard, aber viele Compiler bieten außerdem noch die Funktion `is_heap()` an, die `true` zurückgibt, wenn der Bereich `[first,last)` ein Heap ist, oder `false`, wenn nicht.

Hierzu ein Beispiel, das alle vier Heap-Funktionen im Einsatz zeigt:

```
// stl_algo20.cpp
#include <vector>
#include <algorithm>
#include <iostream>
```

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

int main() {
   vector<int> Vector1;
   vector<int>::iterator Viter;
   // Quelldaten
   int data[]={5,9,3,4,7,5,3,2,1,6};
   int dataSize = sizeof(data)/sizeof(int);
   // in Vector kopieren
   copy( data, data+dataSize, back_inserter(Vector1));

   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << ")" << endl;

   make_heap(Vector1.begin(), Vector1.end());
   cout << "make_heap:  " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << endl;

   // Nächstes Element aus dem Heap herausholen
   pop_heap(Vector1.begin(), Vector1.end());
   Vector1.pop_back();
   cout << "pop_heap :  " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << endl;

   // Ein Element zum Heap hinzufügen
   Vector1.push_back(11);
   push_heap(Vector1.begin(), Vector1.end());
   cout << "push_heap:  " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << endl;

   // Sortieren - anschließend ist Vector1 kein Heap mehr
   sort_heap(Vector1.begin(), Vector1.end());
```

```
   cout << "sort_heap:  " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
   cout << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector1 = ( 5 9 3 4 7 5 3 2 1 6 )
make_heap:  9 7 5 4 6 3 3 2 1 5
pop_heap :  7 6 5 4 5 3 3 2 1
push_heap:  11 7 5 4 6 3 3 2 1 5
sort_heap:  1 2 3 3 4 5 5 6 7 11
```

Nach der Transformation mit `make_heap()` sind die Elemente als Heap sortiert:

9 7 5 4 6 3 3 2 1 5

Diese Umwandlung in einen binären Baum, bei dem der Wert des Knotens immer kleiner oder gleich dem Elternknoten ist, kann man sich so vorstellen:

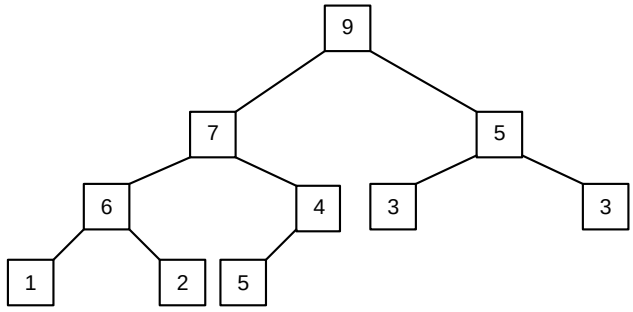

Abbildung 5.2 Anordnung der Elemente nach »make_heap()«

Algorithmen für Minimum und Maximum

Algorithmen	Beschreibung

`template <class T>`
`const T& min(`
`  const T& a, const T& b );`
	Gibt das kleinere von zwei Elementen zurück.

`template <class T,`
`  class Compare>`
`const T& min(`
`  const T& a, const T& b,`
`  Compare comp );`
	Dito, nur wird für den Vergleich `comp` verwendet und nicht der <-Operator.

Tabelle 5.65 Algorithmen für Minimum und Maximum

Algorithmen	Beschreibung

`template <class T>`
`const T& max(`
`  const T& a, const T& b );`
	Gibt das größere von zwei Elementen zurück.

`template <class T,`
`  class Compare>`
`const T& max(`
`  const T& a, const T& b,`
`  Compare comp );`
	Dito, nur wird für den Vergleich `comp` verwendet und nicht der <-Operator.

`template <class ForwardIterator>`
`ForwardIterator min_element(`
`  ForwardIterator first,`
`  ForwardIterator last  );`
	Gibt das kleinste Element aus dem Bereich `[first,last)` zurück.

`template <class ForwardIterator,`
`  class Compare>`
`ForwardIterator min_element(`
`  ForwardIterator first,`
`  ForwardIterator last,`
`  Compare comp);`
	Dito, nur wird für den Vergleich `comp` verwendet und nicht der <-Operator.

`template <class ForwardIterator>`
`ForwardIterator max_element(`
`  ForwardIterator first,`
`  ForwardIterator last );`
	Gibt das größere Element aus dem Bereich `[first,last)` zurück.

`template <class ForwardIterator,`
`  class Compare>`
`ForwardIterator max_element(`
`  ForwardIterator first,`
`  ForwardIterator last,`
`  Compare comp );`
	Dito, nur wird für den Vergleich `comp` verwendet und nicht der <-Operator.

Tabelle 5.65 Algorithmen für Minimum und Maximum (Forts.)

Algorithmen zum lexikografischen Vergleich

Algorithmen	Beschreibung

`template <class InputIterator1,`
`  class InputIterator2>`
`bool lexicographical_compare(`
`  InputIterator1 first1,`
`  InputIterator1 last1,`
`  InputIterator2 first2,`
`  InputIterator2 last2 );`
	Liefert `true` zurück, wenn die Elemente im Bereich `[first1,last1)` lexikografisch kleiner sind als die Elemente im Bereich `[first2,last2)`. Ansonsten wird `false` zurückgegeben.

Tabelle 5.66 Lexikografischer Vergleich

Algorithmen	Beschreibung

`template <class InputIterator1,`
`class InputIterator2,`
`class Compare>`
`bool lexicographical_compare`
`InputIterator1 first1,`
`InputIterator1 last1,`
`InputIterator2 first2,`
`InputIterator2 last2,`
`Compare comp );`
	Dito, nur wird für den Vergleich `comp` verwendet und nicht der `<`-Operator.

Tabelle 5.66 Lexikografischer Vergleich (Forts.)

[»] **Hinweis**

Neben der hier beschriebenen Funktion `lexicographical_compare()` bieten einige Compiler auch noch eine `strcmp()`-ähnliche Alternative mit der Funktion `lexicographical_compare_3way()` an, die einen negativen Wert zurückgibt, wenn der Bereich `[first1,last1)` lexikografisch kleiner ist als `[first2,last2)`. Ein positiver Wert wird zurückgegeben, wenn `[first1,last1)` lexikografisch größer ist als `[first2,last2)`. Sind beide Bereiche gleich, wird null zurückgegeben. Diese Funktion ist allerdings noch nicht im Standard aufgenommen.

Hierzu ein Beispiel, das den lexikografischen Vergleich zweier C-Strings demonstrieren soll:

```
// stl_algo21.cpp
#include <algorithm>
#include <iostream>
#include <iterator>
using namespace std;

int main() {
   char name1[] = "Albert";
   char name2[] = "Adolf";

   cout << "Sortiert (lexikografisch): " << endl;
   if( lexicographical_compare(
          name1, name1+sizeof(name1),
          name2, name2+sizeof(name2)) ) {
      cout << name1 << endl << name2 << endl;
   }
   else {
      cout << name2 << endl << name1 << endl;
   }
```

```
   cout << "Sortiert (lexikografisch - umgekehrt): "
        << endl;
   if( lexicographical_compare(
          name1, name1+sizeof(name1),
          name2, name2+sizeof(name2),
          greater<char>()) ) {
      cout << name1 << endl << name2 << endl;
   }
   else {
      cout << name2 << endl << name1 << endl;
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Sortiert (lexikografisch):
Adolf
Albert
Sortiert (lexikografisch - umgekehrt):
Albert
Adolf
```

Permutation

Eine Permutation (lat.: *permutare* = tauschen) ist eine Veränderung der Reihenfolge von Elementen einer Menge. Man kann sich dies wie bei einem Kartenspiel vorstellen, wo durch das Mischen der Karten diese jedes Mal anders sortiert sind. Dabei handelt es sich immer um eine Permutation der Karten einer Menge.

Beispielsweise versteht man unter einer N-stelligen Permutation die Anordnung einer Menge von N Elementen. So sind zum Beispiel (3, 2, 1), (2, 3, 1) oder (1, 3, 2) drei unterschiedliche Permutationen der Menge { 1, 2, 3 }. Die Anzahl der Elemente aller Permutationen von N Elementen berechnet sich zu N! (Fakultät).

Algorithmus	Beschreibung

`template <class BidirectionalIterator>`
`bool` **`prev_permutation`**`(`
`  BidirectionalIterator first,`
`  BidirectionalIterator last );`
	Führt die vorige Permutation der Elemente im Bereich `[first,last)` aus. Existiert eine Permutation, wird `true` zurückgegeben, ansonsten `false`, wenn die letzte Permutation erreicht wurde. Am Ende sind die Elemente aufsteigend sortiert.

Tabelle 5.67 Permutation

Algorithmus	Beschreibung

`template <`
`  class BidirectionalIterator,`
`  class Compare>`
`bool prev_permutation (`
`  BidirectionalIterator first,`
`  BidirectionalIterator last,`
`  Compare comp );`
	Dito, nur wird für den Vergleich `comp` verwendet und nicht der <-Operator.

`template <`
`  class BidirectionalIterator>`
`bool next_permutation(`
`  BidirectionalIterator first,`
`  BidirectionalIterator last );`
	Führt die nächste Permutation der Elemente im Bereich `[first,last)` aus. Existiert eine Permutation, wird `true` zurückgegeben, ansonsten `false`, wenn die letzte Permutation erreicht wurde. Am Ende sind die Elemente absteigend sortiert.

`template <`
`  class BidirectionalIterator,`
`  class Compare>`
`bool next_permutation(`
`  BidirectionalIterator first,`
`  BidirectionalIterator last,`
`  Compare comp );`
	Dito, nur wird für den Vergleich `comp` verwendet und nicht der <-Operator.

Tabelle 5.67 Permutation (Forts.)

Hierzu die Funktion `next_permutation()` in der Praxis:

```
// stl_algo22.cpp
#include <algorithm>
#include <vector>
#include <iostream>
#include <iterator>
using namespace std;

int main() {
   vector<int> Vector1;
   vector<int>::iterator Viter;
   // Quelldaten
   int data[]={1,2,3};
   int dataSize = sizeof(data)/sizeof(int);
   // in Vector kopieren
   copy( data, data+dataSize, back_inserter(Vector1));

   cout << "Vector1 = ( " ;
   for ( Viter = Vector1.begin( ) ;
         Viter != Vector1.end( ) ; Viter++ )
      cout << *Viter << " ";
```

```
   cout << ")" << endl;

   while( next_permutation(Vector1.begin(),Vector1.end())) {
      cout << "Vector1 = ( " ;
      for ( Viter = Vector1.begin( ) ;
            Viter != Vector1.end( ) ; Viter++ )
         cout << *Viter << " ";
      cout << ")" << endl;
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Vector1 = ( 1 2 3 )
Vector1 = ( 1 3 2 )
Vector1 = ( 2 1 3 )
Vector1 = ( 2 3 1 )
Vector1 = ( 3 1 2 )
Vector1 = ( 3 2 1 )
```

5.3.7 Allokatoren

Auf den vorangegangenen Seiten war schon oft von den Allokatoren die Rede, ohne dass genauer darauf eingegangen wurde. Allokatoren sind ein wesentlicher und wichtiger Bestandteil der STL. Jeder Container verwendet einen und speziell jeder Konstruktor besitzt einen Parameter mit einem Allokator.

Wie auch schon die Iteratoren sind Allokatoren ein abstraktes Konzept. Die Iteratoren wurden dazu verwendet, den Zugriff auf einen Container zu kapseln, damit sich die Algorithmen nicht um den direkten Zugriff kümmern müssen. Ebenso ist es mit den Allokatoren, nur dienen diese zum Kapseln des Zugriffs auf den Speicher. Der Container sollte sich nicht selbst um die Reservierung von Speicherplatz auf dem Heap bemühen, dazu sollte der Allokator verwendet werden. Der Vorteil ist, dass man auf diese Weise die Art der Speicherverwaltung für den Container ändern kann, ohne dass hiervon Implementierungsdetails des Containers betroffen sind.

In Abbildung 5.3 sehen Sie die Zusammenhänge zwischen STL und deren Komponenten. Hierbei lässt sich auch erkennen, dass – bis auf den Iterator – alle Komponenten von STL unabhängig voneinander implementiert werden können, wenn die entsprechenden Schnittstellen eingehalten werden.

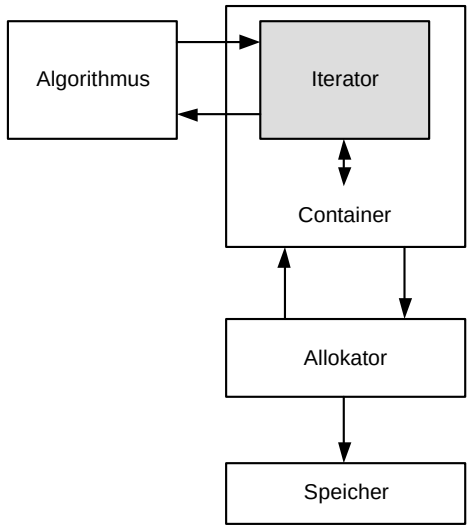

Abbildung 5.3 Aufbau von STL

[»] **Hinweis**

Es ist nicht unbedingt notwendig, bis ins Detail zu verstehen, wie Allokatoren funktionieren – auf den vorangegangenen Seiten ging es schließlich auch ohne. Aber sobald man die Speicherverwaltung von Containern selbst in die Hand nehmen will oder eigene Container entwickelt, kommt man nicht um die Allokatoren herum. Allerdings kann man auch sagen, dass dies selten der Fall sein wird. Trotzdem sind Sie dann für den Fall der Fälle gerüstet und kennen sich zudem besser mit den Interna von STL aus.

Um gleich Missverständnisse aus dem Weg zu räumen: Allokatoren haben nichts mit der dynamischen Speicherreservierung zu tun. Wenn Sie beispielsweise `new T` verwenden, reservieren Sie Speicher für ein neues Objekt vom Typ `T`, und mit `delete ptr` zerstören Sie ein Objekt, auf das der Zeiger `ptr` im Speicher verweist. Wenn Sie jetzt meinen, dass Allokatoren etwas mit `new` und `delete` zu tun haben, irren Sie sich.

Allokatoren verstecken im Grunde nur den Low-Level-Teil der Speicherreservierung von STL-Containern. Daher sollte man auch niemals versuchen, die Methoden von Allokatoren im Code zu verwenden, es sei denn, Sie schreiben einen eigenen Container. Außerdem sollten Sie bei einem eigenen Allokator niemals den Operator `new[]` implementieren.

[»] **Hinweis**

Wenn Sie sich nicht sicher sind, ob und wie Sie Allokatoren einsetzen können, dann sollten Sie diese nicht verwenden.

Der Standard-Allokator

Der Standard-Allokator wird von allen Containern von STL verwendet und ist in der Header-Datei <*memory*> als Standard-Template mit der Bezeichnung `allocator` definiert. Natürlich reserviert `allocator` den Speicherplatz über den Operator `new()`. Hier ein Ausschnitt aus der Header-Datei <*memory*> und dem Standard-Template `allocator`:

```
template<class T>
class allocator {
    // Datentypen von allocator
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef T *pointer;
    typedef const T *const_pointer;
    typedef T& reference;
    typedef const T& const_reference;
    typedef T value_type;

    // Adresse eines Objekts ermitteln
    pointer address(reference val) const;
    // Dito, nur für konstante Allokatoren
    const_pointer address(const_reference val) const;

    template<class Other> struct rebind;
    allocator();
    template<class Other>
     allocator(const allocator<Other>& right);
    template<class Other>
     allocator& operator=(const allocator<Other>& right);
    // Speicher reservieren
    template<class Other>
     pointer allocate(size_type count, const Other *hint=0);
    // Speicher freigeben
    void deallocate(pointer ptr, size_type count);
    // Objekte konstruieren
    void construct(pointer ptr, const Ty& val);
    // Objekte zerstören
    void destroy(pointer ptr);
    // Max. Anzahl Elemente, die verwaltet werden können
    size_type max_size() const;
};

// Vergleichsoperatoren
template<class Typ>
bool operator==(const allocator<Typ> &a1,
```

```
   const allocator<Typ> &a2){
   return(true);
}
template<class Typ>
bool operator!=(const allocator<Typ> &a1,
   const allocator<Typ> &a2){
   return(false);
}
```

Zunächst definiert der Allokator anhand der Datentypen der zu verwaltenden Elemente einige neue Datentypen, die später auch von den Containern übernommen werden:

Datentyp	Beschreibung
`const_pointer`	Typ für konstante Zeiger
`const_reference`	Typ für konstante Referenzen
`difference_type`	Typ für die Angabe von Abständen; normalerweise ein `signed`-Typ
`pointer`	Zeiger-Typ
`reference`	Referenz-Typ
`size_type`	Typ, mit dem Größen ausgedrückt werden; im Allgemeinen ein `unsigned`-Typ
`value_type`	Typ der zu verwaltenden Elemente

Tabelle 5.68 Datentypen von »allocator«

Das Anlegen eines Elements geschieht bei Allokatoren in zwei Stufen. Bei der ersten Stufe wird so viel Speicher angefordert, wie nötig ist, um das Element flach zu kopieren. Jetzt kann in diesem Speicherbereich ein Element konstruiert werden. Der Konstruktor des Elements initialisiert also den Speicher mit entsprechenden Werten und fordert gegebenenfalls vom Element benötigten dynamischen Speicher an.

Die Freigabe von Speicher verläuft wieder umgekehrt. Zuerst wird das Element wieder zerstört, wo der Destruktor aufgerufen wird. Anschließend kann der Speicherbereich vom Allokator freigegeben werden.

Speicher reservieren und freigeben

Zum Reservieren von Speicher verwendet der Allokator folgende Methode:

```
pointer allocate(size_type anz, void* info=0);
```

Diese Methode reserviert Speicher für `anz` Elemente des zu verwaltenden Typs. Der Parameter `info` hat bei Standard-Allokatoren keine Bedeutung und wird gewöhnlich dazu verwendet, bei selbstimplementierten Allokatoren Informatio-

nen zu übergeben, die bei der Vergabe des Speichers berücksichtigt werden. Zur Reservierung des Speichers vom Heap wird auf den Operator `new()` zurückgegriffen. Die Methode `allocate` reserviert zwar den Speicher, aber initialisiert diesen nicht.

Um den Speicher wieder freizugeben, wird die Methode `deallocate` verwendet:

```
void deallocate(pointer ptr, size_type anz);
```

Damit wird für `anz` Elemente reservierter Speicher, beginnend bei der Adresse `ptr`, freigegeben. Hierbei wird der Operator `delete()` verwendet. Wichtig dabei: `deallocate` gibt nur den Speicher frei, ohne das Element zu zerstören.

Elemente konstruieren und zerstören

Wenn der Speicher reserviert wurde, wird dieser durch die Konstruktion eines Objekts initialisiert. Andersherum muss ein konstruiertes Objekt zerstört werden, bevor der Speicher wieder freigegeben wird. Sollte dies nicht der Fall sein, könnten einige Ressourcen, die das Objekt belegt, nicht freigegeben werden. Dadurch entsteht ein Leck (Ressourcenleck).

Für das Konstruieren eines Objekts steht die Methode `construct` zur Verfügung:

```
void construct(pointer ptr, const Typ& obj);
```

Hier initialisieren Sie den Speicher ab der Adresse `ptr` mit dem Objekt `obj`. Dazu wird der Kopierkonstruktor von `Typ` verwendet. Wenn der Speicher bei `ptr` mit dem entsprechenden Allokator reserviert wurde, stimmt die Größe des Speicherbereichs mit der Größe eines Objekts vom Typ `Typ` überein. Natürlich konstruiert `construct` nur ein Objekt in einem bereits vorher reservierten Speicher.

Um das Objekt wieder zu zerstören, wird die Methode `destroy` verwendet:

```
void destroy(pointer ptr);
```

Diese Methode ruft den Destruktor des an der Adresse `ptr` befindlichen Objekts auf. Der Speicher wird aber nicht freigegeben, sondern es wird nur das Objekt zerstört.

Adressen ermitteln und »max_size«

Um die Adresse eines übergebenen Objekts zu erhalten, steht Ihnen die Methode `address` zur Verfügung. Diese Methode existiert in zwei Varianten – eine für veränderbare und eine für konstante Allokatoren:

```
// für variable Allokatoren
pointer address(reference r) const;
```

```
// für konstante Allokatoren
const_pointer address(const_reference r);
```

Wie viele Elemente für einen bestimmten Typ maximal verwaltet werden können, lässt sich mit der Methode `max_size()` ermitteln:

```
size_type max_size() const;
```

Da die Basis die Größe des Datentyps `size_type` ist, können nicht mehr Bytes verwaltet werden als die größte Zahl, die mit einem `size_type` dargestellt werden kann. Wenn diese maximale Anzahl an Bytes noch durch die Größe des zu verwaltenden Datentyps dividiert wird, erhält man den gewünschten Wert.

void-Allokator

Für den Datentyp `void` wurde eine Spezialisierung von `allocator` benötigt und implementiert:

```
template<>
class allocator<void> {
   public:
   typedef void* pointer;
   typedef const void* const_pointer;
   typedef void value_type;
};
```

Vergleichsoperatoren

Die Vergleichsoperatoren == und != vergleichen nicht wirklich zwei Allokator-Instanzen. Da der Allokator nur aus Methoden besteht, müssen zwei Allokator-Instanzen vom selben Typ auch gleich sein. Deswegen wird vom ==-Operator `true` und vom !=-Operator `false` zurückgegeben.

Selbstdefinierten Allokator erstellen und verwenden

Da Sie nun wissen, wie solche Allokatoren in STL implementiert sind, können Sie eine selbstdefinierte Version eines solchen Allokators erstellen. Verwenden können Sie diese mit jedem beliebigen STL- oder auch einem eigenen Container.

[»] **Hinweis**

Es handelt sich hier nicht um eine Anleitung, wie Sie den Standard-Allokator durch den selbstdefinierten Allokator ersetzen können. Solche Maßnahmen sollte man im Grunde sowieso nicht treffen, wenn man nicht genau weiß, was man hier tut. Bei diesem Abschnitt geht es lediglich darum, Ihnen einen Eindruck zu vermitteln, wie STL funktioniert und aufgebaut ist. Daher finden Sie anschließend auch noch ein Beispiel, wie Sie einen eigenen Container à la STL erstellen können, der auch den selbstdefinierten Allokator verwenden kann. Außerdem wird hier nicht im vollen Umfang auf die Allokatoren eingegangen.

Zur Demonstration wird im Folgenden ein einfaches Beispiel verwendet, das statt des Operators `new()` in der Methode `allocate()` und `delete()` in `deallocate()` die C-Funktionen `malloc()` und `free()` aus der Header-Datei <*cstdlib*> benutzen soll. Wir verwenden als Bezeichner für diesen simplen Allokator `mallocator`. Durch diese vereinfachte Verwendung (ohne Speicheroptimierungen) der Speicherverwaltung mit `mallocator` können Sie sich auf die grundlegenden Eigenschaften und Methoden eines selbstdefinierten Allokators konzentrieren.

Hinweis **[«]**

Sie werden feststellen, dass dies ohnehin nur eine Wiederholung des Standard-Allokators darstellt – nur jetzt mit selbstdefiniertem Quellcode.

Mit Blick auf den Standard-Allokator können Sie alle dort definierten Datentypen (Eigenschaften) auch bei unserem eigenen Allokator `mallocator` übernehmen:

```
template <class T> class mallocator {
public:
  typedef T                 value_type;
  typedef value_type*       pointer;
  typedef const value_type* const_pointer;
  typedef value_type&       reference;
  typedef const value_type& const_reference;
  typedef std::size_t       size_type;
  typedef std::ptrdiff_t    difference_type;
};
```

Da Sie die Datentypen des Allokators `mallocator` angegeben haben, können Sie sich an die Arbeit machen und die Methoden schreiben. Hierzu können Sie zunächst die Methode `address` (in der veränderbaren und in der konstanten Version) erstellen, mit der die Adresse eines Objekts ermittelt werden kann:

```
  pointer address(reference x) const {
     return &x;
  }
  const_pointer address(const_reference x) const {
    return &x;
  }
```

Als Nächstes werden wir die Hauptfunktionen `allocate` und `deallocate` erstellen. Wie schon beim Standard-Allokator für Container verwenden wir für `allocate` zwei Parameter. Mit dem ersten Parameter geben Sie an, wie viele Elemente für den zu verwaltenden Typ an Speicher reserviert werden sollen. Der zweite Parameter bleibt auch hier anderen Verwendungen vorbehalten und wird nicht

weiter verwendet. Zur Reservierung des Speichers wird jetzt aber die C-Funktion `malloc()` statt des Operators `new()` eingesetzt.

Ebenso sieht die Syntax unserer eigenen `deallocate`-Methode aus. Auch verwenden wir wie beim Original zwei Parameter mit derselben Bedeutung. Der einzige Unterschied zum Original ist, dass statt des Operators `delete` die C-Funktion `free()` zum Freigeben des Speichers benutzt wird. Hier die beiden selbstdefinierten Methoden `allocate` und `deallocate`:

```
pointer allocate(size_type n, const_pointer = 0) {
  void* p = malloc(n * sizeof(T));
  if (!p)
    throw bad_alloc();
  return static_cast<pointer>(p);
}
void deallocate(pointer p, size_type) {
  free(p);
}
```

[»] Hinweis

In diesem Beispiel wird mit `throw` die Exception `bad_alloc()` geworfen. Die Exceptions werden gesondert in Kapitel 6, »Exception-Handling«, behandelt.

Wie Sie bereits von den Standard-Allokatoren wissen, arbeiten die Methoden `allocate` und `deallocate` nur mit dem Speicher, aber erzeugen bzw. zerstören deswegen noch kein Objekt. Dies wird bei den Allokatoren mit den Methoden `construct` und `destroy` gemacht. Um einen Konstruktor für `T` zu verwenden, genügt es nicht, `new T(val)` anzugeben. Hierzu müssen Sie `new(pointer) T(val)` verwenden. Es ist wichtig, die Adresse mit anzugeben, wo das Objekt konstruiert wird. Diese Adresse wird ebenfalls benötigt, um das Objekt zu zerstören (durch den Destruktor). Hier reicht ebenfalls ein Aufruf von `pointer->~T()` aus. Dazu die beiden Methoden `construct` und `destroy`:

```
void construct(pointer p, const value_type& x) {
  new(p) value_type(x);
}
void destroy(pointer p) {
   p->~value_type();
}
```

Der ein oder andere wird sich jetzt fragen, ob die beiden Methoden `construct` und `destroy` wirklich nötig sind. Sie werden relativ selten verwendet, weil die Speicherreservierung und das Initialisieren häufig auf einmal gemacht werden. Da es aber extrem gefährlich ist, mit Zeigern auf einen uninitialisierten Speicher-

bereich zu hantieren, sollte man diese beiden Methoden immer implementieren. Außerdem gibt es einen Fall, in dem diese Methoden garantiert verwendet werden, und zwar bei der Entwicklung eigener Container-Klassen (was auch noch gezeigt wird).

Da keine dieser Methoden statisch ist, ist auch hier die erste Aufgabe eines Containers, ein Allokator-Objekt zu erzeugen, bevor dieser einen Allokator verwendet. Also benötigen Sie einen Konstruktor. Wenn ein Konstruktor definiert wird, muss logischerweise auch ein Destruktor vorhanden sein. Den Zuweisungsoperator sollten Sie mit einem Kopierschutz (siehe Abschnitt 4.5.5, »Überladen des Zuweisungsoperators«) versehen, damit dieser nicht aus Versehen eingesetzt wird. Die Standard-Allokatoren verwenden hierbei auch keinen Zuweisungsoperator – und wir sind auf der sicheren Seite, weil nicht automatisch einer generiert wird.

```
  // Konstruktoren
  mallocator() {}
  mallocator(const mallocator&) {}
  // Destruktor
  ~mallocator() {}

private:
  // Kopierschutz für Zuweisung!
  void operator=(const mallocator&);
```

Natürlich macht keiner dieser Konstruktoren etwas, da der Allokator gewöhnlich auch keine Eigenschaften zum Initialisieren besitzt.

Hierzu nun die simple Version unseres selbstdefinierten Allokators `mallocator`, bei dem zusätzlich die Methode `max_size()` und die beiden Vergleichsoperatoren == und != implementiert wurden:

```
// mallocator.h
#include <iostream>
#include <cstdlib>
#ifndef MALLOCATOR_H
#define MALLOCATOR_H
using namespace std;

template <class T> class mallocator {
public:
  typedef T                value_type;
  typedef value_type*      pointer;
  typedef const value_type* const_pointer;
  typedef value_type&      reference;
  typedef const value_type& const_reference;
```

```
  typedef size_t       size_type;
  typedef ptrdiff_t    difference_type;

  pointer address(reference x) const {
     return &x;
  }
  const_pointer address(const_reference x) const {
    return &x;
  }

  pointer allocate(size_type n, const_pointer = 0) {
    void* p = malloc(n * sizeof(T));
    if (!p)
      throw bad_alloc();
    return static_cast<pointer>(p);
  }

  void deallocate(pointer p, size_type) {
    free(p);
  }

  size_type max_size() const {
            cout << "max_size: ";
    return static_cast<size_type>(-1) / sizeof(value_type);
  }

  void construct(pointer p, const value_type& x) {
    new(p) value_type(x);
  }
  void destroy(pointer p) { p->~value_type(); }

  // Konstruktoren
  mallocator() {}
  mallocator(const mallocator&) {}
  // Destruktor
  ~mallocator() {}

private:
  // Kopierschutz!
  void operator=(const mallocator&);
};

template <class T>
inline bool operator==(const mallocator<T>&,
                       const mallocator<T>&) {
```

```
  return true;
}

template <class T>
inline bool operator!=(const mallocator<T>&,
                       const mallocator<T>&) {
  return false;
}
#endif
```

Der Einsatz des selbstdefinierten Allokators ist relativ einfach. Sie müssen nur die Argumente der Container-Klasse anpassen. Einen `int`-Vektor verwenden Sie beispielsweise wie folgt:

```
vector<int> intVec;
```

Dieser `int`-Vektor benutzt per Standardeinstellung den Standard-Allokator `std::allocator`. Wenn Sie stattdessen den selbstdefinierten Allokator verwenden wollen, müssen Sie nur Folgendes schreiben:

```
vector<int, mallocator<int> > intVec;
```

Dasselbe gilt selbstverständlich für alle anderen Container-Klassen:

```
list<int, mallocator<int> > intList;
```

Damit verwenden Sie die Container-Klasse `list` mit dem Allokator `mallocator` und nicht mit dem Standard-Allokator. Ein weiteres interessantes Beispiel stellt Folgendes dar:

```
mallocator<int>::pointer valptr;
```

Diese Schreibweise stellt ein Gegenstück zu `int*` dar und lässt sich auch in der Praxis so verwenden. Hierzu ein Programm, das den Einsatz des selbstdefinierten Allokators in der Praxis demonstrieren soll:

```
// mallocator.cpp
#include <iostream>
#include <vector>
#include "mallocator.h"
using namespace std;

class A {
private:
   int ival;
   float fval;
public:
   // Konstruktor
```

```
   A( int i=0, float f=0.0f ): ival(i), fval(f) {}
   // Destruktor
   ~A( ) {}
   void anzeigen() const {
      cout << ival << " : " << fval << "\n";
   }
};

int main() {
   // ein int-Vektor mit 5 Elementen
   vector<int, mallocator<int> > Vectorint(5);
   // ein Vektor der Klasse A mit 3 Elementen
   vector<A, mallocator<A> > VectorA(3);
   // Entspricht einem int*
   mallocator<int>::pointer valptr = new int[10];
   // int-Vektor mit Werten füllen
   for(size_t i = 0; i < Vectorint.size(); ++i)
      Vectorint[i] = i*i;
   // A-Vektor mit Werten füllen
   float f=1.1;
   for(size_t i=0; i < VectorA.size(); ++i) {
      VectorA[i] =  A(i, f);
      f*=2;
   }
   for( size_t i = 0; i < 10; i++) {
      valptr[i] = i+i;
   }

   // int-Vektor dynamisch vergrößern (Zahl 1111 anhängen)
   Vectorint.insert(Vectorint.end(), 1111);
   // A-Vektor um ein Element erweitern
   // (111, 111.111 anhängen)
   VectorA.insert(VectorA.end(), A(111, 111.111));
   // ... hinten anhängen geht auch so ...
   VectorA.push_back( A(222, 222.222));

   // Einzelne Elemente ausgeben über ...
   // ... die Benutzung als Array
   cout << "int-Vektor: \n";
   for(size_t i = 0; i < Vectorint.size(); ++i)
      cout << Vectorint[i] << ", ";
   cout << endl << "A-Vektor: \n";
   for(size_t i = 0; i < VectorA.size(); ++i)
      VectorA[i].anzeigen();
   cout << "valptr: \n";
   for(size_t i = 0; i < 10; i++)
```

```
      cout << valptr[i] << " ";
   cout << endl;

   // Zugriff auf einzelne Elemente über den Iterator
   cout << "\nint-Vektor (mit Iterator): \n";
   for(vector<int, mallocator<int> >::const_iterator
          myiter=Vectorint.begin();
       myiter != Vectorint.end(); ++myiter )
      cout << *myiter << endl;
   cout << "A-Vektor (mit Iterator): \n";
   for( vector<A, mallocator<A> >::const_iterator
           myiter = VectorA.begin();
        myiter != VectorA.end(); ++myiter )

      myiter->anzeigen();
   cout << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
int-Vektor:
0, 1, 4, 9, 16, 1111,
A-Vektor:
0 : 1.1
1 : 2.2
2 : 4.4
111 : 111.111
222 : 222.222
valptr:
0 2 4 6 8 10 12 14 16 18

int-Vektor (mit Iterator):
0
1
4
9
16
1111
A-Vektor (mit Iterator):
0 : 1.1
1 : 2.2
2 : 4.4
111 : 111.111
222 : 222.222
```

Eine eigene Container-Klasse schreiben

Jetzt haben Sie gesehen, wie Sie STL um einen eigenen Allokator erweitern könnten – auch wenn das Beispiel nicht unbedingt zu empfehlen ist. Ebenso wie den Allokator können Sie auch STL um eigene Container-Klassen erweitern. Dabei können Sie entweder den Standard-Allokator von STL oder auch den selbstdefinierten verwenden.

Natürlich kann hier nur oberflächlich auf die Erstellung eigener Container-Klassen eingegangen werden. Container-Klassen wie `vector` oder `map` sind relativ kompliziert und lassen sich nicht auf ein paar Seiten oder mit einem Listing erklären. Als Beispiel erstellen wir eine einfache Klasse namens `Array`, in der die Anzahl der Elemente mit dem Konstruktor angegeben werden und nicht geändert werden können. Das ist nicht sehr nützlich, aber einfach zu verstehen. Für die Parameter des Templates benötigen Sie zwei Typen, und zwar den Typ des Elements und den Allokator-Typ.

Container, die einen Allokator verwenden, benutzen auch die in Allokatoren eingebauten Datentypen: `value_type`, `reference`, `const_reference`, `size_type`, `difference_type`, `iterator` und `const_iterator`. Gewöhnlich werden diese Typen direkt von den Allokatoren des Containers verwendet. Dies ist übrigens auch ein Grund, warum Sie bei den selbstdefinierten Allokatoren diese Datentypen (siehe Tabelle 5.68) ebenfalls implementieren sollten.

Die Iteratoren einer STL-Klasse sind kein Teil des Allokators, sondern eine eigene Art von Klasse, die eng mit den Containern verbunden ist. Da unser Beispiel relativ einfach ist und die Elemente in einem einfachen Speicherblock gespeichert werden, verwenden wir als Iterator einen Zeiger auf den Anfang und einen auf das Ende des `Array`-Speicherblocks.

Was jetzt beim Grundgerüst einer Container-Klasse noch fehlt, sind ein Konstruktor und ein Destruktor. Hierbei müssen Sie sich nun darüber Gedanken machen, wie Sie mit dem Allokator umgehen. Es soll schließlich auch möglich sein, dass der Anwender gegebenenfalls einen selbstdefinierten Allokator implementiert (und nicht nur den Standard-Allokator verwenden kann). Das ist zwar gewöhnlich nicht der Fall, aber ein STL-Container bietet dieses Feature ebenfalls an (und der Standard erwartet dies auch).

Der Konstruktor von `Array` initialisiert die Allokator-(Basis-)Klasse, holt sich einen ganzen Speicherblock (`allocate`), der groß genug ist für `n` Elemente, und initialisiert die einzelnen Elemente mit einem Wert (`construct`). Sobald eines der Elemente beim Initialisieren (`construct`) fehlschlägt, wird eine Exception ausgelöst und alles wieder rückgängig gemacht.

Hinweis [«]

Die Exceptions werden in Kapitel 6, »Exception-Handling«, behandelt.

Der Destruktor von `Array` ist einfacher zu implementieren. Wird dieser ausgeführt, werden zunächst die einzelnen Elemente zerstört (`destroy`), und anschließend wird der Speicherplatz freigegeben (`deallocate`) – die umgekehrte Reihenfolge wie beim Konstruktor.

Sie haben mit diesem Container lediglich ein Grundgerüst erstellt. In unserem Beispiel geht es ja auch nur darum, den Konstruktor und den Destruktor einer Container-Klasse zu erstellen und dabei den Allokator zu verwenden.

Hinweis [«]

Der C++-Standard schreibt vor, was ein solcher Container benötigt, damit er komplett ist und dem Standard entspricht. Diese Angaben finden Sie im Standard in der Tabelle 64 in 23.1. In der Suchmaschine lässt sich dies auch mit dem Suchbegriff »lib.container.requirements« finden:

```
// array.h
#ifndef ARRAY_H
#define ARRAY_H
#include <iostream>
using namespace std;

template <class T, class Allocator = std::allocator<T> >
class Array : private Allocator {
public:
  typedef T value_type;

  typedef typename Allocator::reference reference;
  typedef typename Allocator::const_reference
          const_reference;

  typedef typename Allocator::size_type size_type;
  typedef typename Allocator::difference_type
          difference_type;

  typedef typename Allocator::pointer       iterator;
  typedef typename Allocator::const_pointer const_iterator;

  typedef Allocator allocator_type;
  allocator_type get_allocator() const {
    return static_cast<const Allocator&>(*this);
  }
```

```
  iterator begin()             { return first; }
  iterator end()               { return last; }
  const_iterator begin() const { return first; }
  const_iterator end() const   { return last; }

  Array(size_type n = 0,
        const T& x = T(),
        const Allocator& a = Allocator());
  Array(const Array&);
  ~Array();

  Array& operator=(const Array&);

private:
  typename Allocator::pointer first;
  typename Allocator::pointer last;
};

// Konstruktor
template <class T, class Allocator>
Array<T, Allocator>::Array(size_type n, const T& x,
                           const Allocator& a)
                    : Allocator(a), first(0), last(0) {
  if (n != 0) {
    first = Allocator::allocate(n);
    size_type i;
    try {
      for (i = 0; i < n; ++i) {
        Allocator::construct(first + i, x);
      }
    }
    catch(...) {
      for(size_type j = 0; j < i; ++j) {
        Allocator::destroy(first + j);
      }
      Allocator::deallocate(first, n);
      throw;
    }
  }
}

// Destruktor
template <class T, class Allocator>
Array<T, Allocator>::~Array() {
  if (first != last) {
```

```
    for (iterator i = first; i < last; ++i)
      Allocator::destroy(i);
    Allocator::deallocate(first, last - first);
  }
}
#endif
```

Jetzt können Sie die einfache `Array`-Container-Klasse verwenden. Allerdings lassen sich hierbei nur mit `Array<T>` oder `Array<T, Allocator>` neue Objekte eines bestimmten Typs `T` und (optional) mit dem Allokator `Allocator` erzeugen. Mehr wurde hier nicht implementiert. Verwenden können Sie das erstellte Beispiel standardmäßig mit dem Standard-Allokator und einmal – explizit – mit dem selbstdefinierten Allokator `mallocator` wie folgt:

```
// myContainer.cpp
#include <iostream>
#include <vector>
#include "mallocator.h"
#include "array.h"
using namespace std;

class A {
private:
   int ival;
   float fval;
public:
   // Konstruktor
   A( int i=0, float f=0.0f ): ival(i), fval(f) {}
   // Destruktor
   ~A( ) {}
   void anzeigen() const {
      cout << ival << " : " << fval << "\n";
   }
};

int main() {
   // Verwendet mallocator als Allokator
   Array<int, mallocator<int> > Arrint(5);
   // Verwendet mallocator als Allokator
   Array<A, mallocator<A> > ArrA(3);
   // Verwendet den Standard-Allokator
   Array<int> Arrint2(10);
   // Verwendet den Standard-Allokator
   Array<A> ArrA2(5);
   return 0;
}
```

Mit dem Exception-Handling (Ausnahmebehandlung) haben Sie die Möglichkeit, auf »unerwartete« Aktionen eines Codeabschnitts zu reagieren. Gewöhnlich sind solche Aktionen Fehlersituationen, die beim Programm auftreten können.

6 Exception-Handling

In einem Programm können laufend irgendwelche »unerwarteten« Fehler auftreten, die den Programmablauf stören bzw. das Programm unstabil machen. Beispiele dafür sind:

- mangelnder Speicher
- falscher Zugriff auf Dateien
- Fehler bei arithmetischen Berechnungen
- Zugriff auf unerlaubte Adressen (Stichwort: Zeiger)
- falsche Eingabe bei der Standardeingabe des Anwenders
- Fehler auf dem System

Bisher dürften Sie auf Fehler mit einer der drei folgenden Möglichkeiten reagiert haben:

- Programm abbrechen – mit dieser Möglichkeit behandeln Sie zwar den Fehler, aber häufig ist ein Beenden des Programms nicht wünschenswert.
- Fehler ignorieren (und hoffen, dass es schon gutgehen wird) – die wohl schlechteste, aber leider häufig praktizierte Möglichkeit.
- Einen eigenen Fehlercode verwenden – hierbei werden Funktionen auf dessen Rückgabewert überprüft, und es wird in einen entsprechenden Fehlercode verzweigt. Allerdings lässt sich dies nicht bei Konstruktoren anwenden, weil diese keinen Rückgabewert zurückgeben.

Von den hier erwähnten Möglichkeiten erscheint die dritte am vernünftigsten, aber es ist nicht möglich, so auf alle Ausnahmen zu reagieren, die bei einer Software auftreten. Zudem führt dies zu einem recht unübersichtlichen Code, weil Fehlerbehandlungen und der normale Code miteinander vermischt werden.

6.1 Exception-Handling in C++

Das Prinzip der Behandlung von *Exceptions* (Ausnahmen) ist im Grunde nicht kompliziert. Hierzu gibt es sogenannte *Exception-Handles*, die nicht Teil des Programms sind und nur dazu verwendet werden, auf Ausnahmesituationen zu reagieren. Wenn eine solche Ausnahme eintritt, lösen Sie als Programmierer eine Exception aus. Dadurch wird die aktuelle Ausführung des Programms unterbrochen und mit der Behandlung der Exception und dem Handle fortgefahren (man trennt immer das Programm und das Behandeln einer Exception voneinander).

[»] **Hinweis**

Es ist nicht sonderlich hilfreich, den Begriff *Handle* ins Deutsche zu übersetzen, weil sich ein »Ausnahme-Handgriff« nun mal nicht so gut und verständlich anhört wie ein »Exception-Handle«.

Dieser Handle behandelt jetzt die Ausnahme. Nach der Ausführung des Exception-Handles kann mit dem eigentlichen Programmablauf fortgefahren werden. Der Vorteil dieses Verfahrens bei einem Fehler ist, dass auf diese Weise Fehler viel zentraler behandelt werden können.

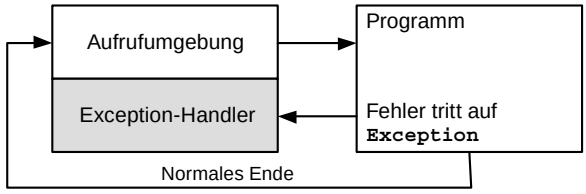

Abbildung 6.1 Konzept der Ausnahmebehandlung von C++

6.2 Eine Exception auslösen

Wenn im Programm ein Fehler aufgetreten ist, kann dieser mit einer `throw`-Anweisung an die Aufrufumgebung gemeldet werden, also eine Exception ausgelöst werden. Mit folgender Syntax können Sie ein Fehlerobjekt an die Aufrufumgebung »werfen«:

```
throw FehlerObjekt;
```

Natürlich kann das Objekt ein beliebiger Basisdatentyp oder eine Klasse sein, mit Ausnahme von `void`. Bei Zeigern muss man außerdem beachten, dass diese immer auf statische oder globale Objekte verweisen.

Auch wenn man einen beliebigen Typ verwenden kann, werden in der Praxis gewöhnlich eigene Fehlerklassen definiert. Eine solche Fehlerklasse enthält im Allgemeinen die Komponente, die zur Behandlung von Fehlern genaue Informationen über den Fehler zur Verfügung stellt.

Wird also eine Ausnahme mit `throw` ausgelöst, dann wird die Programmausführung unterbrochen (wie in Abschnitt 6.1, »Exception-Handling in C++«, beschrieben) und mit dem Exception-Handle fortgefahren.

Man sollte nicht den Fehler machen, die `throw`-Anweisung mit alten `goto`-Konstrukten zu vergleichen, die beispielsweise früher in C verwendet wurden. Bei einer `throw`-Anweisung werden, im Gegensatz zu einer Sprunganweisung wie `goto`, alle lokalen Objekte freigegeben (auch bekannt als *Stack-Unwinding*). Dadurch ist garantiert, dass beim Auslösen einer Exception keine Speicherlecks und sonstiger Datenmüll übrigbleiben.

6.3 Eine Exception auffangen – Handle einrichten

Nachdem man eine Exception mit `throw` ausgelöst hat, benötigt man einen Handle, der das Fehlerobjekt auffängt und diese Ausnahme behandelt. Auch wenn kein entsprechender Handle eingerichtet wurde, verliert sich das Programm nicht in einem undefinierbaren Zustand, sondern wird mit der Standardfunktion `terminate()` ordnungsgemäß beendet.

Für die Behandlung einer Exception benötigen Sie Folgendes:

- den Programmteil, in dem eine Exception ausgelöst wird
- den Handle – zur Behandlung der gegebenenfalls verschiedenen Typen von Exceptions

Für diesen Zweck gibt es die beiden Schlüsselwörter `try` und `catch`, die folgende Syntax besitzen:

```
try {
   // Codeabschnitt, der eine Exception auslösen kann
}
catch( ExceptionTyp Bezeichner1 ) {
   // Handle für Fehler vom Typ ExceptionTyp
}
```

Im `try`-Block wird zunächst festgelegt, welche Exceptions »aufgefangen« werden können. Es werden natürlich nur die Exception-Typen aufgefangen, für die auch ein Handle definiert wurde. Ein solcher Handle wird durch eine dem `try`-Block folgende `catch`-Anweisung definiert. Dem `try`-Block muss immer mindestens

eine `catch`-Anweisung folgen. Natürlich können noch weitere Handles mit `catch` definiert werden. Allerdings müssen sich die einzelnen `catch`-Handles durch ihre Parameter unterscheiden:

```
try {
   // Codeabschnitt, der eine Exception auslösen kann
}
catch( ExceptionTyp1 Bezeichner ) {
   // Handle für Fehler vom Typ ExceptionTyp1
}
catch( ExceptionTyp2 Bezeichner ) {
   // Handle für Fehler vom Typ ExceptionTyp2
}
...
// ggf. weitere Exception-Handle
```

Hierzu ein einfaches Beispiel, das eine Funktion `funktion()` mit einem Integer-Wert als Parameter im `try`-Block aufruft. Je nach übergebenem Wert wird eine bestimmte Exception ausgelöst oder, sofern der Wert `1` übergeben wurde, das Programm ordnungsgemäß ausgeführt, ohne dass eine Exception ausgelöst wird:

```
// exceptions1.cpp
#include <iostream>
using namespace std;

class myExceptionClass {
   // ...
};

void funktion( int ival );

int main() {
   int val, ret_val=0;
   cout << "Bitte Eingabe machen (-1,0,1): ";
   cin >> val;

   try {
      funktion( val );
      // wenn dies ausgegeben wurde, keine Exception
      cout << "Keine Exception ausgelöst" << endl;
   }
   catch( const char* msg ) {
      cout << "Exception-Handle für const char* "
           << "wurde aufgerufen\n"
           << "->( " << msg << " )<-" << endl;
```

```
      ret_val = 0;
   }
   catch( myExceptionClass& ) {
      cout << "Exception-Handle für myExceptionClass "
           << "wurde aufgerufen\n";
      ret_val = 1;
   }
   return ret_val;
}

void funktion( int ival ) {
   cout << "funktion() aufgerufen" << endl;
   switch ( ival ) {
      case -1: myExceptionClass e;
               throw e;
               // break nicht nötig
      case  0: throw "Programmende";
               // break bei throw nicht mehr nötig
      case  1: cout << "funktion() wird ordnungsgemäß"
                    << " ausgeführt" << endl;
               break;
      default: throw "Unbekannter Fehler";
   }
   cout << "funktion() ordnungsgemäß beendet" << endl;
}
```

Das Programm bei der Ausführung:

```
Bitte Eingabe machen (-1,0,1): -1
funktion() aufgerufen
Exception-Handle für myExceptionClass wurde aufgerufen

Bitte Eingabe machen (-1,0,1): 0
funktion() aufgerufen
Exception-Handle für const char* wurde aufgerufen
->( Programmende )<-

Bitte Eingabe machen (-1,0,1): 1
funktion() aufgerufen
funktion() wird ordnungsgemäß ausgeführt
funktion() ordnungsgemäß beendet
Keine Exception ausgelöst

Bitte Eingabe machen (-1,0,1): 5
funktion() aufgerufen
```

```
Exception-Handle für const char* wurde aufgerufen
->( Unbekannter Fehler )<-
```

Im Beispiel wurde bei der `throw`-Anweisung

```
case -1: myExceptionClass e;
         throw e;
```

das Fehlerobjekt zunächst definiert, bevor es »geworfen« wurde. Es ist allerdings auch möglich, das Fehlerobjekt direkt bei der `throw`-Anweisung zu verwenden:

```
// ... Alternativ:
case -1: throw myExceptionClass();
```

Des Weiteren sollte hier erwähnt werden, dass wir in diesem Beispiel das Fehlerobjekt nicht verwenden bzw. mit keiner Funktionalität versehen haben, weshalb hier auch `catch(myExceptionClass&)` statt `catch(myExceptionClass& e)` benutzt wurde.

6.3.1 Reihenfolge (Auflösung) der Ausnahmen

Wie man im Beispiel *exceptions1.cpp* sehen kann, können immer mehrere `catch`-Handles eingerichtet werden. Da sich diese durch ihre Parameter unterscheiden müssen, kann man Ähnlichkeiten zur Funktions-Überladung erkennen.

Auf der Suche nach dem passenden Handle, werden die einzelnen `catch`-Handles von oben nach unten durchlaufen, um zu ermitteln, welcher Handle der passende ist. Dies ist der Fall, wenn

- die Exception und der im Handle angegebene Parameter vom selben Typ sind.
- der im Handle angegebene Parameter eine direkte oder indirekte Basisklasse der Exception ist.
- der angegebene Parameter ebenso wie die Exception ein Zeigertyp ist, der durch eine Standardumwandlung umgewandelt werden kann.

Wenn keiner dieser `catch`-Handles passt, wird die Funktion `terminate()` aufgerufen, die das Programm regulär beendet.

6.3.2 Alternatives »catch(...)«

Wenn eine unbekannte Exception ausgelöst wurde bzw. deren Behandlung nicht so wichtig erscheint, ist die Standardlösung, das Programm einfach mit `terminate()` zu beenden, nicht immer eine zufriedenstellende Wahl. Für solche Zwecke wurde ein alternatives `catch` eingeführt, das alle ausgelösten Exceptions einfängt, für die noch kein Handle eingerichtet wurde. Dieses alternative `catch`

enthält als Parameter drei Punkte (was als *Ellipse* bezeichnet wird), steht für alle Ausnahmen und passt auch auf jeden Typ.

Gerade weil diese `catch`-Anweisung mit der Ellipse für jeden Typ passt, muss sie unbedingt als letzter Handle angegeben werden, weil sonst ein definierter Exception-Handle dahinter niemals ausgeführt würde, sondern immer die Alternative für das passende `catch(...)`. Hierzu das Beispiel *exceptions1.cpp*, erweitert durch die alternative `catch`-Anweisung:

```
// exceptions2.cpp
#include <iostream>
using namespace std;

class myExceptionClass {
   // ...
};

void funktion( int ival );

int main() {
   int val, ret_val=0;
   cout << "Bitte Eingabe machen (-1,0,1): ";
   cin >> val;

   try {
      funktion( val );
      // wenn dies ausgegeben wurde, keine Exception
      cout << "Keine Exception ausgelöst" << endl;
   }
   catch( const char* msg ) {
      cout << "Exception-Handle für const char* "
           << "wurde aufgerufen\n"
           << "->( " << msg << " )<-" << endl;
      ret_val = 0;
   }
   catch( myExceptionClass& ) {
      cout << "Exception-Handle für myExceptionClass "
           << "wurde aufgerufen\n";
      ret_val = 1;
   }
   catch( ... ) {
      cout << "Eine Exception unbekannten Typs wurde"

           << " ausgelöst" << endl;
   }
```

```
   return ret_val;
}

void funktion( int ival ) {
   cout << "funktion() aufgerufen" << endl;
   switch ( ival ) {
      case -1: myExceptionClass e;
               throw e;
               // break nicht nötig
      case  0: throw "Programmende";
               // break bei throw nicht mehr nötig
      case  1: cout << "funktion() wird ordnungsgemäß"
                    << " ausgeführt" << endl;
               break;
      default: throw 1; // Ruft catch( ... ) auf
   }
   cout << "funktion() ordnungsgemäß beendet" << endl;
}
```

Das Programm bei der Ausführung:

```
Bitte Eingabe machen (-1,0,1): 5
funktion() aufgerufen
Eine Exception unbekannten Typs wurde ausgelöst
```

6.3.3 Stack-Abwicklung (Stack-Unwinding)

Natürlich stellt sich die Frage, was mit den »Speicherleichen« bei einer Exception passiert. Wenn eine Exception ausgelöst wird, wird der aktuelle `try`-Block verlassen. Wenn dies geschieht, wird aber zunächst nicht, wie vielleicht zu vermuten wäre, zum passenden `catch`-Handle gesprungen, sondern es werden zuerst die Veränderungen des Stacks, die seit dem Eintritt in den `try`-Block vorgenommen wurden, rückgängig gemacht. Dies betrifft insbesondere die lokalen und nicht statischen Objekte, die innerhalb des `try`-Blocks definiert wurden. Dieser Abbau des Stacks wird *Stack-Unwinding* genannt. Erst nach dem Abbau des Stacks wird zum passenden `catch`-Handle gesprungen.

Das Ganze lässt sich relativ einfach mit dem Listing *exceptions2.cpp* demonstrieren. Hier brauchen Sie nur noch eine neue Klasse mit Konstruktor und Destruktor zu erstellen und davon ein Objekt im `try`-Block zu definieren:

```
// exceptions3.cpp
#include <iostream>
using namespace std;
class myExceptionClass {
```

```
   //...
};

class myObject {
public:
   myObject() {
      cout << "Konstruktor für myObject" << endl;
   }
   ~myObject() {
      cout << "Destruktor für myObject" << endl;
   }
};

void funktion( int ival );

int main() {
   int val, ret_val=0;
   cout << "Bitte Eingabe machen (-1,0,1): ";
   cin >> val;

   try {
      myObject obj1;
      funktion( val );
      // wenn dies ausgegeben wurde, keine Exception
      cout << "Keine Exception ausgelöst" << endl;
   }
   catch( const char* msg ) {
      cout << "Exception-Handle für const char* "
           << "wurde aufgerufen\n"
           << "->( " << msg << " )<-" << endl;
      ret_val = 0;
   }
   catch( myExceptionClass& ) {
      cout << "Exception-Handle für myExceptionClass "
           << "wurde aufgerufen\n";
      ret_val = 1;
   }
   catch( ... ) {
      cout << "Eine Exception unbekannten Typs wurde"
           << " ausgelöst" << endl;
   }
   return ret_val;
}
void funktion( int ival ) {
   cout << "funktion() aufgerufen" << endl;
```

```
   switch ( ival ) {
      case -1: myExceptionClass e;
               throw e;
               // break nicht nötig
      case  0: throw "Programmende";
               // break bei throw nicht mehr nötig
      case  1: cout << "funktion() wird ordnungsgemäß"
                       " ausgeführt" << endl;
               break;
      default: throw 1; // Ruft catch( ... ) auf
   }
   cout << "funktion() ordnungsgemäß beendet" << endl;
}
```

Das Programm bei der Ausführung:

```
Bitte Eingabe machen (-1,0,1): -1
Konstruktor für myObject
funktion() aufgerufen
Destruktor für myObject
Exception-Handle für myExceptionClass wurde aufgerufen
```

An der Ausgabe können Sie erkennen, dass erst die Aufräumarbeiten des Stacks gemacht werden, bevor der `catch`-Handle seine Arbeit verrichtet.

6.3.4 try-Blöcke verschachteln

Wenn es (unbedingt) nötig ist, kann man auch mehrere `try`-Blöcke ineinander verschachteln:

```
try {
     // Hier kann eine Exception ausgelöst werden
     try {
        // Hier kann eine weitere Exception ausgelöst werden
     }
     catch( ExceptionT1 e ) {
        // Dieser catch-Handle gilt nur für Exceptions,
        // die vom inneren try-Block ausgelöst werden
     }
catch( ExceptionT2 e ) {
   // Dieser Handle fängt Exception vom äußeren
   // try-Block auf
}
```

Das Verschachteln soll an unserem Beispiel *exceptions3.cpp* demonstriert werden:

```
// exceptions4.cpp
#include <iostream>
using namespace std;

class myExceptionClass { };

void funktion( int ival );

int main() {
   int val, val2, ret_val=0;
   cout << "Bitte Eingabe machen (-1,0,1): ";
   cin >> val;

   try {
      funktion( val );
      // wenn dies ausgegeben wurde, keine Exception
      cout << "Keine Exception ausgelöst" << endl;
      try {
         cout << "Noch eine Eingabe machen: ";
         cin >> val2;
         funktion( val2 );
      }
      catch( myExceptionClass& ) {
      cout << "Innerer Exception-Handle für"
           << " myExceptionClass "
           << "wurde aufgerufen\n";
      ret_val = 1;
      }
      catch( ... ) {
         cout << "Eine Exception unbekannten Typs"
              << " wurde INNEN ausgelöst" << endl;
      }
   }
   catch( const char* msg ) {
      cout << "Exception-Handle für const char* "
           << "wurde aufgerufen\n"
           << "->( " << msg << " )<-" << endl;
      ret_val = 0;
   }
   catch( myExceptionClass& ) {
      cout << "Äußerer Exception-Handle für"
           << " myExceptionClass "
           << "wurde aufgerufen\n";
      ret_val = 1;
   }
```

```
   catch( ... ) {
      cout << "Eine Exception unbekannten Typs wurde"
           << " ausgelöst" << endl;
   }
   return ret_val;
}

void funktion( int ival ) {
   cout << "funktion() aufgerufen" << endl;
   switch ( ival ) {
      case -1: myExceptionClass e;
               throw e;
               // break nicht nötig
      case  0: throw "Programmende";
               // break bei throw nicht mehr nötig
      case  1: cout << "funktion() wird ordnungsgemäß"
                    << " ausgeführt" << endl;
               break;
      default: throw 1; // Ruft catch( ... ) auf
   }
   cout << "funktion() ordnungsgemäß beendet" << endl;
}
```

Das Programm bei der Ausführung:

```
Bitte Eingabe machen (-1,0,1): 1
funktion() aufgerufen
funktion() wird ordnungsgemäß ausgeführt
funktion() ordnungsgemäß beendet
Keine Exception ausgelöst
Noch eine Eingabe machen: 1
funktion() aufgerufen
funktion() wird ordnungsgemäß ausgeführt
funktion() ordnungsgemäß beendet

Bitte Eingabe machen (-1,0,1): -1
funktion() aufgerufen
Äußerer Exception-Handle für myExceptionClass wurde aufgerufen

Bitte Eingabe machen (-1,0,1): 1
funktion() aufgerufen
funktion() wird ordnungsgemäß ausgeführt
funktion() ordnungsgemäß beendet
Keine Exception ausgelöst
Noch eine Eingabe machen: 0
```

```
funktion() aufgerufen
Eine Exception unbekannten Typs wurde INNEN ausgelöst

Bitte Eingabe machen (-1,0,1): 1
funktion() aufgerufen
funktion() wird ordnungsgemäß ausgeführt
funktion() ordnungsgemäß beendet
Keine Exception ausgelöst
Noch eine Eingabe machen: -1
funktion() aufgerufen
Innerer Exception-Handle für myExceptionClass wurde aufgerufen
```

Hier lässt sich schon erkennen, dass der innere `try`-Block nicht mehr ausgeführt wird, wenn der äußere `try`-Block schon eine Exception auslöst.

6.3.5 Exception weitergeben

Beim Betrachten des Listings *exceptions4.cpp* fällt auf, dass der innere Block eigentlich dasselbe macht wie der äußere. Beide »Blöcke« besitzen gleichwertige Handles. Hier kann man die Exception weitergeben bzw. nochmals auslösen. Dies erledigt man mit einem einfachen `throw` ohne sonstige Angaben. Damit werfen Sie die Exception aus einem `catch`-Handle heraus, so dass sie von einem umgebenden `try`-Block weiterbehandelt werden kann. Solch ein »Wiederrauswerfen« ist allerdings nur innerhalb eines `catch`-Handle sinnvoll:

```
try {
     // Hier kann eine Exception ausgelöst werden
     try {
        // Hier kann eine weitere Exception ausgelöst werden
     }
     catch( ExceptionT1 e ) {
        // Damit wird die Exception an den äußeren
        // catch-Handle weitergegeben
        throw;
     }
catch( ExceptionT2 e ) {
   // Dieser Handle fängt Exceptions vom äußeren und jetzt
   // auch inneren try-Block auf
}
```

Umgeschrieben auf das Listing *exceptions4.cpp* sieht der entscheidende Codeausschnitt wie folgt aus:

```
// exception5.cpp
...
```

```
int main() {
   int val, val2, ret_val=0;
   cout << "Bitte Eingabe machen (-1,0,1): ";
   cin >> val;

   try {
      funktion( val );
      // wenn dies ausgegeben wurde, keine Exception
      cout << "Keine Exception ausgelöst" << endl;
      try {
         cout << "Noch eine Eingabe machen: ";
         cin >> val2;
         funktion( val2 );
      }
      catch( myExceptionClass& ) {
         // Weitergeben an den äußeren Handle
         throw;
      }
      catch( ... ) {
         cout << "Eine Exception unbekannten Typs wurde"
              << " INNEN ausgelöst" << endl;
      }
   }
   catch( const char* msg ) {
      cout << "Exception-Handle für const char* "
           << "wurde aufgerufen\n"
           << "->( " << msg << " )<-" << endl;
      ret_val = 0;
   }
   catch( myExceptionClass& ) {
      cout << " Exception-Handle für myExceptionClass "
           << "wurde aufgerufen\n";
      ret_val = 1;
   }
   catch( ... ) {
      cout << "Eine Exception unbekannten Typs wurde"
           << " ausgelöst" << endl;
   }
   return ret_val;
}
```

Da im verschachtelten `try`-Block das Gleiche nochmals behandelt wird und im Beispiel innen nicht auf `catch(const char*)` reagiert wird, könnte man auch den inneren `catch`-Handle auf `catch(...)` beschränken und mit `throw` alle Exceptions gleich an die äußeren Handles (genauer an den `try`-Block) weiterschicken:

```
// exception6.cpp
...
int main() {
   int val, val2, ret_val=0;
   cout << "Bitte Eingabe machen (-1,0,1): ";
   cin >> val;

   try {
      funktion( val );
      // wenn dies ausgegeben wurde, keine Exception
      cout << "Keine Exception (außen) ausgelöst" << endl;
      try {
         cout << "Noch eine Eingabe machen: ";
         cin >> val2;
         funktion( val2 );
         cout << "Keine Exception (innen) ausgelöst"
              << endl;
      }
      catch( ... ) {
         throw;
      }
   }
   catch( const char* msg ) {
      cout << "Exception-Handle für const char* "
           << "wurde aufgerufen\n"
           << "->( " << msg << " )<-" << endl;
      ret_val = 0;
   }
   catch( myExceptionClass& ) {
      cout << "Exception-Handle für myExceptionClass "
           << "wurde aufgerufen\n";
      ret_val = 1;
   }
   catch( ... ) {
      cout << "Eine Exception unbekannten Typs wurde"
           << " ausgelöst" << endl;
   }
   return ret_val;
}
```

Hiermit geben Sie alle Exceptions, die im inneren `try`-Block auftreten, an den äußeren Block weiter.

[»] **Hinweis**

Im Prinzip ist das Beispiel hier nicht besonders sinnvoll, da man wohl kaum `try`-Blöcke verschachtelt, die dieselbe Arbeit bzw. Überprüfung verrichten. Das ist so, als würde man zweimal hintereinander in einem `try`-Block die Eingabe überprüfen. In diesem Beispiel geht es jedoch nur um das Verständnis, dass und wie man `try`-Blöcke verschachteln kann und dass es möglich ist, Exceptions wieder aus einem `catch`-Handle hinauszuwerfen.

6.4 Ausnahmeklassen (Fehlerklassen)

Die Fehlerklassen, die Sie bisher verwendet haben, hatten keine Funktionalität. Eine Klasse wird als Fehlerklasse bezeichnet, wenn dieser Typ eines Objekts ist, der nur im Falle eines Fehlers erzeugt wird. Natürlich kann eine solche Klasse, wie andere Klassen auch, Eigenschaften (Daten) und Methoden (Funktionen) besitzen. In der Regel verwendet man solche Klassen, um dem Anwender Informationen zum aufgetretenen Fehler zu übermitteln.

Hierzu soll das Beispiel *Array1.cpp* aus dem Abschnitt 5.2.4, »Weitere Template-Parameter«, verwendet werden. Zunächst wird eine Fehlerklasse `Overflow` definiert. Findet eine Bereichsüberschreitung (Buffer Overflow) in der Klasse `Array` statt, wird die Fehlerklasse `Overflow` mit dem fehlerhaften Index erzeugt und als Exception ausgeworfen. Der entsprechende Exception-Handle greift dann über eine Methode auf die Nummer des fehlerhaften Indexes zurück und gibt diese auf dem Bildschirm aus. Im Beispiel lösen wir absichtlich im `try`-Block eine Bereichsüberschreitung aus:

```
// Array2.cpp
#include <iostream>
using namespace std;

class Overflow {
   private:
      int range;
   public:
      Overflow( int i ) : range(i) {}
      int get_range() const { return range; }
};

template <class T, int n>
class Array {
private:
   T array[n];
```

```
public:
   int get_size () const {
      return n;
   }
   T& operator[] (int i);
};

template <class T, int n>
T &Array<T, n>::operator[] (int i) {
   if(i < 0 || i >= n ) {
      throw Overflow(i);
   }
   return array[i];
}

int main( void ) {
   Array<char, 20> str1;
   Array<int, 10> int1;

   cout << "Wie heißen Sie: ";
   cin.getline( &str1[0], str1.get_size() );
   cout << "Hallo ";
   for(int i=0; str1[i] != '\0'; i++)
      cout << str1[i];
   cout << endl;

   try {
      // Absichtlicher Überlauf
      str1[20] = 'A';
   }
   catch( Overflow& e1 ) {
      cout << "Überlauf an Pos. " << e1.get_range() << endl;
   }

   try {
      // Wieder eine Bereichsüberschreitung
      for( int i = 0; i < 20; i++ ) {
         int1[i] = i*i;
         cout << "int1[" << i << "] = " << int1[i] << endl;
      }
   }
   catch( Overflow& e2 ) {
      cout << "Überlauf an Pos. " << e2.get_range() << endl;
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Wie heißen Sie: Jürgen Wolf
Hallo Jürgen Wolf
Überlauf an Pos. 20

int1[0] = 0
int1[1] = 1
int1[2] = 4
int1[3] = 9
int1[4] = 16
int1[5] = 25
int1[6] = 36
int1[7] = 49
int1[8] = 64
int1[9] = 81
Überlauf an Pos. 10
```

6.4.1 Klassenspezifische Exceptions

Wenn Sie eine Fehlerklasse schreiben, die nur für eine bestimmte Klasse zum Einsatz kommt, können Sie diese auch innerhalb der Klasse definieren. Sie müssen dabei nur beachten, dass die Fehlerklasse im `public`-Bereich der Klasse definiert wird. Der Zugriff auf die Fehlerklasse innerhalb einer Klasse erfolgt dann über den Scope-Operator. Die Verwendung von klassenspezifischen Exceptions hat zum einen den Vorteil, dass keine Namenskonflikte mit anderen Klassen auftreten, und zum anderen können die Eigenschaften (Daten) einer Klasse berücksichtigt werden und müssen dem Konstruktor nicht extra mit übergeben werden.

Angewandt auf das Beispiel *Array2.cpp*, sieht der Einsatz klassenspezifischer Exceptions wie folgt aus:

```
// Array3.cpp
#include <iostream>
using namespace std;

template <class T, int n>
class Array {
private:
   T array[n];
public:
   class Overflow {
   private:
         int range;
   public:
```

```
        Overflow( int i ) : range(i) {}
        int get_range() const { return range; }
        // Möglich, da jetzt klassenspezifisch
        int get_maxN() const { return n-1; }
   };
   int get_size () const {
      return n;
   }
   T& operator[] (int i);
};

template <class T, int n>
T &Array<T, n>::operator[] (int i) {
   if(i < 0 || i >= n ) {
      throw Overflow(i);
   }
   return array[i];
}

int main( void ) {
   Array<char, 20> str1;
   Array<int, 10> int1;

   try {
      // Absichtlicher Überlauf
      str1[20] = 'A';
   }
   catch( Array<char,20>::Overflow& e1 ) {
      cout << "Überlauf an Pos. " << e1.get_range() << endl;
      cout << "Zulässig sind: " << e1.get_maxN() << endl;
   }

   try {
      // Wieder eine Bereichsüberschreitung
      for( int i = 0; i < 20; i++ ) {
         int1[i] = i*i;
         cout << "int1[" << i << "] = " << int1[i] << endl;
      }
   }
   catch( Array<int, 10>::Overflow& e2 ) {
      cout << "Überlauf an Pos. " << e2.get_range() << endl;
      cout << "Zulässig sind: " << e2.get_maxN() << endl;
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Überlauf an Pos. 20
Zulässig sind: 19

int1[0] = 0
int1[1] = 1
int1[2] = 4
int1[3] = 9
int1[4] = 16
int1[5] = 25
int1[6] = 36
int1[7] = 49
int1[8] = 64
int1[9] = 81
Überlauf an Pos. 10
Zulässig sind: 9
```

6.5 Standard-Exceptions

C++ stellt für die Behandlung von Ausnahmen auch einige Standard-Exception-Klassen zur Verfügung. Alle diese Klassen sind von der Basisklasse `exception` abgeleitet und liegen ebenfalls im Namensraum `std`. Hierzu die Hierarchie der Standardfehlerklassen im Überblick:

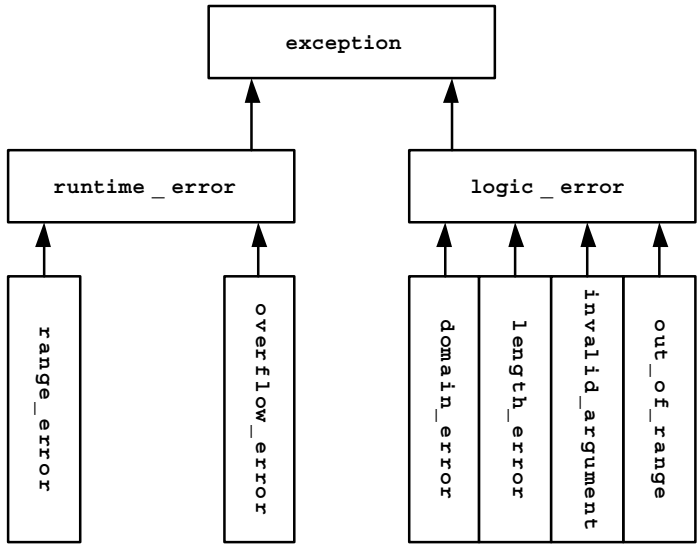

Abbildung 6.2 Vererbung der Standardfehlerklassen

Ausgehend von der Basisfehler-Klasse `exception` unterscheidet man im Grunde zwischen zwei Arten von Fehlern:

- Laufzeitfehler (`runtime_error`) – das sind die Fehler, die nicht mehr während der Kontrolle des Programms auftreten. Solche Fehler können jederzeit auftreten und nicht verhindert werden. Dies kann zum Beispiel durch eine defekte Hardware verursacht werden oder durch einen Überlauf bei einer arithmetischen Berechnung. Zu solchen Laufzeitfehlern gehören die Klassen `range_error`, `overflow_error` und `underflow_error`.
- Logische Fehler (`logic_error`) – hierbei handelt es sich um vermeidbare Fehler, deren Ursachen auf einer logischen Erklärung basieren, meistens ein Fehler im Programmablauf. Zu solchen logischen Fehlern gehören `domain_error`, `invalid_argument`, `length_error` und `out_of_range`.

Um diese Exceptions der Standardbibliothek zu verwenden, müssen Sie die Header-Datei <*stdexcept*> mit einbinden. Anschließend stehen Ihnen diese Fehlerklassen im Namensbereich `std` zur Verfügung.

6.5.1 Virtuelle Methode »what()«

In der Basisklasse `exception` ist eine virtuelle Methode `what()` definiert, die allen Standard-Exceptions zur Verfügung steht. Diese Methode gibt einen String zurück, der die Meldung der Fehlerursache enthält. Dieser Text wird in der Regel vom `catch`-Handle an die Standardfehlerausgabe geschickt. Die Syntax der Methode `what()` sieht folgendermaßen aus:

```
virtual const char* what (void) const;
```

6.5.2 Anwenden der Standard-Exceptions

Die Standardfehlerklassen können wie selbstdefinierte Fehlerklassen verwendet werden. Wenn Sie eine Exception auslösen, geben Sie einen Nachrichten-String mit an, der dem `catch`-Handle dann über die Methode `what()` zur Verfügung steht (siehe Abschnitt 6.5.1, »Virtuelle Methode 'what()'«):

```
throw out_of_range("Bereichsüberschreitung");
```

Diese Exception können Sie nun mit dem Exception-Handle wie folgt verarbeiten:

```
try {
   // Bereichsüberschreitung machen
}
catch( out_of_range& e1 ) {
```

```
   cout << "Bereichsüberschreitung bei "
        << e1.what() << endl;
}
```

out_of_range (logic_error)

Die Exception `out_of_range` dient der Auslösung einer Exception, wenn ein Wert nicht mehr in einem gültigen Bereich (Adresse) liegt. Beispielsweise verwenden die Standardklassen `bitset` (siehe STL) oder `string` diese Exception, wenn ein Index benutzt wird, der außerhalb des zulässigen Bereichs liegt. Natürlich lässt sich `out_of_range` auch für den eigenen Bedarf verwenden. Hierzu ein Beispiel:

```
// stdexcept1.cpp
#include <iostream>
#include <stdexcept>
#include <bitset>
using namespace std;

template <class T, int n>
class Array {
private:
   T array[n];
public:
   int get_size () const {
      return n;
   }
   T& operator[] (int i);
};

template <class T, int n>
T &Array<T, n>::operator[] (int i) {
   if(i < 0 || i >= n ) {
      throw out_of_range("Bereichsüberschreitung");
   }
   return array[i];
}

int main( void ) {
   Array<int, 10> int1;
   bitset<4> bits;

   try {
      // Bereichsüberschreitung
      bits.set(5);
```

```
    }
    catch( out_of_range& e1 ) {
       cout << "Exception ausgelöst von "
            << e1.what() << endl;
    }

    try {
       // Wieder eine Bereichsüberschreitung
       for( int i = 0; i < 20; i++ ) {
          int1[i] = i*i;
          cout << "int1[" << i << "] = " << int1[i] << endl;
       }
    }
    catch( out_of_range& e2 ) {
       cout << e2.what() << endl;
    }

    try {
       string str1("Jürgen ");
       string str2("Wolf");
       str1.append(str2, 7, 3 );
       cout << str1 << endl;
    }
    catch ( exception &e3 ) {
       cerr << "Exceptions ausgelöst von: " << e3.what( )
            << endl;
    };
    return 0;
}
```

Das Programm bei der Ausführung:

```
Exception ausgelöst von bitset::set
int1[0] = 0
int1[1] = 1
int1[2] = 4
int1[3] = 9
int1[4] = 16
int1[5] = 25
int1[6] = 36
int1[7] = 49
int1[8] = 64
int1[9] = 81
Bereichsüberschreitung
Exceptions ausgelöst von: basic_string::append
```

invalid_argument (logic_error)

Diese Exception wird ausgelöst, wenn ein falsches Argument verwendet wurde. Standardmäßig wird auch diese Exception von den Klassen `bitset`, `deque`, `string` und `vector` verwendet. Im folgenden Beispiel soll ein `string` in ein `bitset` umgewandelt werden. Dabei enthält ein String ein anderes Zeichen als `0` und `1` und daher ein `invalid_argument` (ungültiges Argument).

```
// stdexcept2.cpp
#include <iostream>
#include <stdexcept>
#include <bitset>
using namespace std;

int main( void ) {
   try {
      // Ok ...
      bitset<32> bitset1( string("10101010101010"));
      // Fehler ...
      bitset<8> bitset2( string("1111000 5"));
   }
   catch ( invalid_argument &e1 ) {
      cerr << "Exception ausgelöst von " << e1.what( )
           << endl;
   }
   catch( out_of_range& e2 ) {
      cerr << "Exception ausgelöst von " << e2.what( )
           << endl;
   }
   catch( ... ) {
      cerr << "Exception unbekannter Herkunft" << endl;
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Exception ausgelöst von bitset::_M_copy_from_string.
```

length_error (logic_error)

Tritt diese Exception auf, wird angezeigt, dass ein Objekt erzeugt wurde, das größer ist, als maximal erlaubt. Dies kann beispielsweise der Fall sein, wenn versucht wird, einen String über die maximale Länge (`max_size()+1`) mit Zeichen zu befüllen. Bei den heutigen Compilern dürfte man recht selten an diese Kapazitätsgrenze stoßen (ca. eine Milliarde Zeichen), aber was nicht unmöglich ist,

sollte man niemals ausschließen. Standardmäßig kann diese Exception von der Klasse `string` oder `vector` geworfen werden.

domain_error (logic_error)

Innerhalb der Standardbibliothek wird diese Exception nicht eingesetzt. Sie wird dazu verwendet, eine Exception innerhalb eines Anwendungsbereiches (Bereichsfehler) auszulösen.

ios_base::failure (logic_error)

Diese Exception kann von Ein- und Ausgabeklassen (`istream`, `ostream` usw.) geworfen werden und muss explizit aktiviert werden. Beispielsweise im Fall von `cin`:

```
// Exception aktiviert
cin.exceptions (ios::eofbit | ios::failbit);
```

Die Auslösung einer Exception durch Streams wird also mit der Methode `exceptions()` freigegeben. Als Parameter können Sie eines oder – getrennt durch ein bitweises Oder – mehrere der folgenden Flags verwenden:

Flag	Beschreibung
`ios::failbit`	Die erwarteten Zeichen konnten nicht eingelesen werden (bei der Eingabe), oder die Zeichen konnten bei der Ausgabe nicht geschrieben werden.
`ios::eofbit`	Das Dateiende wurde beim Einlesen erreicht.
`ios::badbit`	Die Daten des Ein- bzw. Ausgabe-Streams sind nicht mehr korrekt.

Tabelle 6.1 Mögliche Flag-Exceptions für Streams

Wenn eine Exception abgefangen wurde, können Sie zum Beispiel bei `cin` mit

```
cin.exceptions()
```

das Bit-Muster des Fehlers ermitteln. Hierzu ein einfaches Beispiel, das das zuvor Beschriebene anhand von `cin` in der Praxis demonstriert:

```
// stdexcept3.cpp
#include <iostream>
#include <stdexcept>
using namespace std;

int main( void ) {
   int val;
   // Exception für cin aktivieren
   cin.exceptions (ios::eofbit | ios::failbit);
```

```
   try {
      cout << "Bitte Eingabe machen (Integer): ";
      cin >> val;
   }
   catch(ios_base::failure& e) {
     //...
     cout << "Exception bei cin ... " << endl;
     cout << "Bitmuster des Fehlers: "
          << cin.exceptions() << endl;
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Bitte Eingabe machen (Integer): a
Exception bei cin ...
Bitmuster des Fehlers: 6
```

[»] **Hinweis**

Diese Erweiterung von `ios_base::failure` wurde erst später in den C++-Standard aufgenommen, weshalb es sein kann, dass sie nicht bei allen Compilern vorhanden ist.

»overflow_error« und »underflow_error« (»runtime_error«)

Fehlerobjekte der Klasse `overflow_error` bzw. `underflow_error` werden ausgelöst, wenn bei einer Berechnung ein arithmetischer Überlauf bzw. Unterlauf aufgetreten ist. In der Standardbibliothek macht davon nur die Klasse `bitset` Gebrauch (zumindest vom Überlauf).

range_error (runtime_error)

Dieses Fehlerobjekt wird ausgelöst, wenn eine Bereichsüberschreitung stattfindet. Diese wird beispielsweise durch die `at()`-Methoden ausgelöst.

6.6 System-Exceptions

Die Standardbibliothek selbst verwendet auch einige Exceptions, die von der Sprache selbst geworfen werden. Auch hier sind alle Standardausnahmen von der Basisklasse `exception` abgeleitet.

6.6.1 bad_alloc

`bad_alloc` wird ausgeworfen, wenn der Operator `new()` einen angeforderten Speicher nicht reservieren konnte. Diese Exception-Klasse ist allerdings in der Header-Datei <*new*> definiert:

```
// stdexcept4.cpp
#include <iostream>
#include <new>
using namespace std;

int main( void ) {
   int *valPtr;
   try {
      valPtr = new int[10];
   }
   catch(bad_alloc& e) {
     //...
     cout << "Exception bei new... " << endl;
     cout << e.what() << endl;
   }
   return 0;
}
```

6.6.2 bad_cast

Diese Exception wird beim `dynamic_cast`-Operator (siehe Abschnitt 3.7.2, »Explizite Typumwandlung«) ausgelöst, wenn die Typumwandlung fehlschlägt. Die `bad_cast`-Exception-Klasse ist in der Header-Datei <*typeinfo*> definiert.

6.6.3 bad_typeid

Diese Exception wird durch den `typeid`-Operator (siehe Abschnitt 7.4, »Typerkennung zur Laufzeit«) ausgelöst, wenn eine Typinformation zu einem dereferenzierten `NULL`-Zeiger ermittelt werden soll. Auch diese Exception-Klasse ist in der Header-Datei <*typeinfo*> definiert.

6.6.4 bad_exception

Diese Exception wird ausgelöst, wenn die Ausnahmebehandlung selbst Probleme bekommt. Dies kann beispielsweise der Fall sein, wenn eine Methode oder Funktion eine Exception auslöst, die nicht in der Exception-Spezifikation der Methoden bzw. Funktionen aufgeführt wurde. Diese Exception-Klasse ist in der Header-Datei <*exception*> definiert.

6.7 Exception-Spezifikation

Bei Exceptions, die von einer Funktion ausgelöst werden, ist es wichtig zu wissen, welche Exceptions durch diese Funktion ausgelöst werden können, weil diese zu den Eigenschaften der Funktion gehören. Für solche Zwecke haben Sie in C++ die Möglichkeit, dies bei der Deklaration der Funktion anzugeben. Man spricht hierbei von einer *Exception-Spezifikation*, die einer Funktionsdeklaration angehängt wird:

```
void funktion( char* ) throw( runtime_error );
```

Natürlich kann man auch eine ganze Liste von Exceptions festlegen, die aus der Funktion an die Aufrufumgebung weitergegeben wird:

```
void funktion(char*) throw(runtime_error,myerr,myerrClass);
```

Somit können von dieser Funktion aus die Exceptions `runtime_error`, `myerr` und `myerrClass` in die Aufrufumgebung geworfen werden. Bezogen auf das Beispiel *exceptions4.cpp*, sieht diese Funktion mit der Exception-Spezifikation wie folgt aus:

```
void funktion( int ival )
throw(myExceptionClass, const char*, int) {
   cout << "funktion() aufgerufen" << endl;
   switch ( ival ) {
      case -1: myExceptionClass e;
               throw e;
               // break nicht nötig
      case  0: throw "Programmende";
               // break bei throw nicht mehr nötig
      case  1: cout << "funktion() wird ordnungsgemäß"
                    << " ausgeführt" << endl;
               break;
      default: throw 1; // Ruft catch( ... ) auf
   }
   cout << "funktion() ordnungsgemäß beendet" << endl;
}
```

Wenn Sie keine Exception-Spezifikation angeben, bedeutet dies, dass alle Exceptions zulässig sind, die von der Funktion in die Aufrufumgebung geworfen werden.

Eine leere Exception-Liste hingegen wie beispielsweise

```
// Hiermit sind keine Exceptions zulässig
void funktion( int ival ) throw()
```

erlaubt keine Exception. Das heißt, jede Exception, die in einer solchen Spezifikation ausgelöst wird, ist unzulässig und ruft `unexpected()` (siehe nächsten Abschnitt) auf.

Wenn Sie die Basisklasse eines Exception-Typs aufrufen, sind alle abgeleiteten Klassen zugelassen. Geben Sie zum Beispiel `logic_error` an, könnte die Funktion die Fehlerklassen `domain_error`, `invalid_argument`, `length_error` und `out_of_range` auslösen – alles wäre zulässig.

6.7.1 Unerlaubte Exceptions

Es wurde bereits im letzten Abschnitt erwähnt, dass `unexpected()` aufgerufen wird, wenn eine Exception ausgelöst wird, die nicht in der Spezifikation angegeben wurde. Standardmäßig bedeutet dies gleichzeitig das Ende des Programms, weil `unexpected()` wiederum die Funktion `terminate()` aufruft.

Allerdings können Sie auch eingreifen und einen `unexpected`-Handle einrichten, der immer dann aufgerufen wird, wenn eine Exception ausgelöst wurde, die nicht der Exception-Spezifikation entspricht, womit `unexpected()` aufgerufen wird. Diesen Handle können Sie mit der Funktion `set_unexpected()` einrichten. Die Syntax dieser Funktion lautet:

```
#include <exception>
...
 // Der Handle
void expectHandle( ) {
   // Code für den Handle
}
...
// Handle installieren
set_unexpected( expectHandle );
```

Wenn bereits die Header-Datei <*stdexcept*> eingebunden wurde, muss die Header-Datei <*exception*> nicht mehr extra eingebunden werden.

Falls jetzt eine unerwartete Exception ausgelöst wurde und Sie diese mit einem Handle abgefangen haben, können Sie entweder das Programm mit einem Aufruf von `terminate()`, `exit()` bzw. `abort()` beenden, oder Sie werfen erneut eine zulässige Exception, die in der Spezifikation vorkommt. Hierzu ein Beispiel, das das Einrichten und das Eintreten einer unerwarteten Exception demonstriert:

```
// unexception.cpp
#include <iostream>
using namespace std;
```

```
class myExceptionClass {
   // ...
};

void funktion( int ival )
   throw( myExceptionClass, const char* );
void expectHandle( );

int main() {
   int val, ret_val=0;

   set_unexpected( expectHandle );

   cout << "Bitte Eingabe machen (-1,0,1): ";
   cin >> val;

   try {
      funktion( val );
      // wenn dies ausgegeben wurde, keine Exception
      cout << "Keine Exception ausgelöst" << endl;
   }
   catch( const char* msg ) {
      cout << "Exception-Handle für const char* "
           << "wurde aufgerufen\n"
           << "->( " << msg << " )<-" << endl;
      ret_val = 0;
   }
   catch( myExceptionClass& ) {
      cout << "Exception-Handle für myExceptionClass "
           << "wurde aufgerufen\n";
      ret_val = 1;
   }
   return ret_val;
}

void funktion( int ival )
throw( myExceptionClass, const char* ) {
   cout << "funktion() aufgerufen" << endl;
   switch ( ival ) {
      case -1: myExceptionClass e;
               throw e;
               // break nicht nötig
      case  0: throw "Programmende";
               // break bei throw nicht mehr nötig
```

```
      case  1: cout << "funktion() wird ordnungsgemäß"
                    << " ausgeführt" << endl;
               break;
      default: throw 99; // Unerwartete Exception
   }
   cout << "funktion() ordnungsgemäß beendet" << endl;
}

void expectHandle( ) {
   cout << "Eine unerwartete Exception ist eingetreten ..."
        << endl;
   throw "Unerwartetes Ende";
}
```

Das Programm bei der Ausführung:

```
Bitte Eingabe machen (-1,0,1): 5
funktion() aufgerufen
Eine unerwartete Exception ist eingetreten ...
Exception-Handle für const char* wurde aufgerufen
->( Unerwartetes Ende )<-
```

Wenn erneut im `unexpected`-Handle eine Exception ausgelöst wird, die nicht der Spezifikation entspricht, wird das Programm mit `terminate()` beendet. Dieses Beenden können Sie allerdings abfangen, indem Sie die Standard-Exception `bad_exception` abfangen, die hierbei ausgelöst wird (oder aber Sie richten einen `terminate`-Handle ein).

6.7.2 terminate-Handle einrichten

Die Funktion `terminate()` wird immer dann aufgerufen, wenn das Exception-Handling nicht ordnungsgemäß abgelaufen ist. Mögliche Gründe hierfür wären unter anderem:

- Für die Exception gibt es keinen passenden Handle.
- Bei der Implementierung des Exception-Handles ist ein Fehler aufgetreten.
- Während des Stack-Unwindings ruft ein Destruktor eine weitere Exception auf.
- Bei der Fehlerklasse ruft der Kopierkonstruktor eine Exception auf.

Wollen Sie jetzt, dass in diesen und anderen Fällen die Funktion `terminate()` das Programm nicht gleich beendet (`terminate()` ruft standardmäßig die Funktion `abort()` auf), können Sie einen `terminate`-Handle einrichten. Dies geschieht mit der Funktion `set_terminate()`. Die Syntax und auch die Verwendung von

`set_terminate()` und `set_unexpected()`sind gleich, beide besitzen dieselbe Schnittstelle:

```
#include <exception>
...
 // Der Handle
void terminateHandle( ) {
   // Code für den Handle
}
...
// Handle installieren
set_terminate( terminateHandle );
```

Hierzu ein Beispiel, das ähnlich wie schon beim Listing *unexception.cpp* ausgeführt wird:

```
// terminate.cpp
#include <iostream>
using namespace std;

class myExceptionClass {
   // ...
};

void funktion( int ival )
throw( myExceptionClass, const char* );
void terminateHandle( );

int main() {
   int val, ret_val=0;
   terminate_Handle BackupHandle;
   cout << "Bitte Eingabe machen (-1,0,1): ";
   cin >> val;

   try {
      BackupHandle = set_terminate( terminateHandle );
      funktion( val );
      // Standard-Handle wieder verwenden
      set_terminate(BackupHandle);
      // wenn dies ausgegeben wurde, keine Exception
      cout << "Keine Exception ausgelöst" << endl;
   }
   catch( const char* msg ) {
      cout << "Exception-Handle für const char* "
           << "wurde aufgerufen\n"
           << "->( " << msg << " )<-" << endl;
```

```
      ret_val = 0;
   }
   catch( myExceptionClass& ) {
      cout << "Exception-Handle für myExceptionClass "
           << "wurde aufgerufen\n";
      ret_val = 1;
   }
   return ret_val;
}

void funktion( int ival ) throw( myExceptionClass,
                                 const char* ) {
   cout << "funktion() aufgerufen" << endl;
   switch ( ival ) {
      case -1: myExceptionClass e;
               throw e;
               // break nicht nötig
      case  0: throw "Programmende";
               // break bei throw nicht mehr nötig
      case  1: cout << "funktion() wird ordnungsgemäß"
                    << " ausgeführt" << endl;
               break;
      default: throw 99; // Unerwartetes Ende
   }
   cout << "funktion() ordnungsgemäß beendet" << endl;
}

void terminateHandle( ) {
   cout << "Ein unerwartetes Ende ist eingetreten ..."
        << endl;
}
```

Das Programm bei der Ausführung:

```
Bitte Eingabe machen (-1,0,1): 6
funktion() aufgerufen
Ein unerwartetes Ende ist eingetreten ...
```

Neben der STL hat die C++-Standardbibliothek noch viel mehr zu bieten. Zum einen die String-Bibliothek, die das Verwenden von Zeichenketten (im Gegensatz zu »char«) zum Kinderspiel macht. Zum anderen gibt es für die Ein-/Ausgabe eine sehr umfangreiche Bibliothek im C++-Standard. Außerdem bietet der Standard auch für komplexe mathematische Berechnungen einige sehr interessante numerische Bibliotheken an. Und für die Erkennung von Typen während der Laufzeit stellt Ihnen C++ ebenfalls eine Möglichkeit zur Verfügung. All dies soll in diesem Kapitel zusammengefasst werden.*

7 C++-Standardbibliothek

7.1 Die String-Bibliothek (string-Klasse)

In Abschnitt 2.4, »Zeichenketten (C-Strings) – char-Array« – haben Sie bereits erfahren, dass es in C++ keinen eigenen Datentyp zur Darstellung von Strings gibt. Zu diesem Zweck wurden (und werden intern auch in der `string`-Klasse) Arrays vom Typ `char` verwendet.

Zur Verarbeitung dieser C-Strings stehen in der C-Standardbibliothek zahlreiche Funktionen zur Verfügung. Diese Funktionen und Utilities für C-Strings sind in den folgenden Header-Dateien enthalten:

<*cctype*>, <*cwtype*>, <*cstring*>, <*cwchar*>, <*cstdlib*>

Ein Problem all dieser Funktionen war allerdings, dass keine Kontrollmechanismen für die C-String-Verarbeitung vorhanden waren. Dinge wie die Speicherverwaltung und Pufferüberläufe (Buffer Overflow) mussten vom Programmierer selbst verwaltet werden. Natürlich kann diese Eigenverantwortung des Programmierers auch ein erhebliches Plus an Geschwindigkeit darstellen. Aber die Vergangenheit hat gezeigt, dass sich durch Pufferüberläufe immer wieder Sicherheitslöcher zum System öffnen ließen.

In C++ wurde daher eine Bibliothek für Strings implementiert, die auf Klassen basiert. Alle Deklarationen dieser String-Bibliothek sind in der Header-Datei <*string*> (nicht zu verwechseln mit der C-Bibliothek `<cstring>`) enthalten.

Außerdem sind sämtliche Typdefinitionen, Instantiierungen, globale Funktionen und Operatoren im Namensbereich `std` enthalten.

[»] **Hinweis**

Zwar wird die String-Bibliothek gesondert in einem Abschnitt behandelt, dennoch ist sie auch Teil der STL – eben eine echte Container-Klasse.

Die String-Bibliothek erlaubt einen komfortableren Umgang mit Strings. Zusätzlich ist hier auch eine Speicherverwaltung implementiert, Pufferüberlaufe gehören damit der Vergangenheit an.

Die String-Bibliothek definiert hierfür den Typ `basic_string`, der ein Klassen-Template ist und das Arbeiten mit verschiedenen Arten von Zeichen erlaubt. Standardmäßig existieren hier Instanzen für `char` und `wchar_t` (*wide character type*). So ist die Klasse `string` eine Spezialisierung für `char` und `wstring` eine Spezialisierung für den Datentyp `wchar_t`.

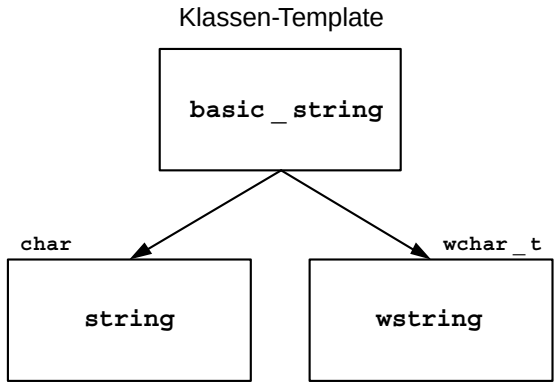

Abbildung 7.1 Spezialisierung der Klasse »basic_string«

Die Syntax der Klasse `basic_string` sieht folgendermaßen aus:

```
namespace std {
     template<class charT,
              class traits = char_traits<charT>,
              class Allocator = allocator<charT> >
     class basic_string;
}
```

Die Bedeutung der drei Template-Parameter ist folgende:

- Der erste Parameter `charT` ist der Datentyp des Zeichens (meistens `char` oder `wchar_t`).

- Beim zweiten Parameter handelt es sich um ein Klassen-Template `traits`, das standardmäßig ebenfalls mit dem Parameter `charT` belegt wird. Mit dem Klassen-Template `traits` kann man die Eigenschaften des Zeichentyps `charT` festlegen. Dies sind Dinge wie das Stringende-Zeichen oder der Vergleich zweier Zeichen.
- Der letzte Parameter ist ein Allokator, der ebenfalls mit einem Standardwert (dem Standard-Allokator) belegt wird. Ein Allokator (siehe Abschnitt 5.3.7, »Allokatoren«) legt die Datentypen und Methoden zur Speicherverwaltung fest.

Die Syntax der beiden Spezialisierungen von `char` alias `string` und `wchar_t` alias `wstring` sieht folgendermaßen aus:

```
namespace std {
   typedef basic_string<char> string;
}

namespace std {
   typedef basic_string<wchar_t> wstring;
}
```

7.1.1 Exception-Handling

Die Methoden des Klassen-Templates `basic_string` können folgende zwei Arten von Exceptions auslösen:

Exception	Beschreibung
`out_of_range`	Eine unzulässige Positionsangabe in einem String wurde durchgeführt.
`length_error`	Der String lässt sich aufgrund einer zu großen Länge nicht mehr darstellen.

Tabelle 7.1 Mögliche Exceptions des Klassen-Templates »basic_string«

7.1.2 Datentypen

Die auf den folgenden Seiten beschriebenen Schnittstellen der Klasse `basic_string` beziehen sich nur auf die Spezialisierung von `string`. Die von `string` bereitgestellten öffentlichen Datentypen sind folgende:

Datentyp	Bedeutung
`string::value_type`	Der Typ der Zeichen; für den Typ `string` ist dies beispielsweise `char` (gleichwertig zu `traits_type::char_type`).

Tabelle 7.2 Öffentliche Datentypen des Klassen-Templates »basic_string«

Datentyp	Bedeutung
`string::reference`	Referenz auf ein Zeichenelement; für den Typ `string` ist dies beispielsweise `char&`.
`string::const_reference`	Dito, aber nur lesend verwendbar.
`string::pointer`	Ein Zeiger auf ein Zeichen; für den Typ `string` ist dies beispielsweise `char*`.
`string::const_pointer`	Dito, aber nur lesend verwendbar.
`string::traits_type`	Der Typ des `traits`-Zeichens, das als zweiter Parameter der Klasse `basic_string` angegeben wurde. Für den Typ `string` ist dies beispielsweise `char_traits <char>`.
`string::size_type`	Der Typ für die Darstellung von Positions- und Längenangaben, der als ganzzahliger vorzeichenloser Typ interpretiert wird. Für `string` ist dies beispielsweise `size_t`.
`string::iterator`	Der Typ des Iterators; für `string` ist dies beispielsweise `char*` (siehe auch Abschnitt 5.3.4, »Iteratoren«).
`string::const_iterator`	Dito, aber kann nur lesend verwendet werden.
`string::reverse_iterator`	Reverser Iterator; auch als `const`-Version vorhanden.

Tabelle 7.2 Öffentliche Datentypen des Klassen-Templates »basic_string« (Forts.)

7.1.3 Strings erzeugen (Konstruktoren)

Zur Erzeugung von Strings stehen mehrere Konstruktoren (siehe Tabelle 7.3) zur Verfügung. Jeder dieser Konstruktoren reserviert den nötigen Speicherplatz und kopiert anschließend gegebenenfalls die als Argument übergebenen Zeichen in den aktuellen String.

Methode	Beschreibung
`string();`	Erzeugt einen String mit der Länge 0.

`string( const string& str,`
`size_type pos = 0,`
`size_type n = npos );`
	Erzeugt einen String mit `n` Zeichen aus dem String `str` ab der Position `pos`. Dabei wird höchstens bis zum letzten Zeichen von `str` kopiert.

`string( const char* str );`	Erzeugt einen String aus dem C-String `str`.
`string( const char* str,` `size_type n );`	Erzeugt einen String aus den ersten `n` Zeichen des C-Strings `str`.
`string( size_type n, char c );`	Erzeugt einen String mit `n` Zeichen, wobei die `n` Zeichen mit dem Zeichen `c` befüllt werden.
`~string();`	Destruktor; zerstört den String wieder.

Tabelle 7.3 Konstruktoren und Destruktoren der Klasse »string«

Hierzu ein Beispiel, das all diese Konstruktoren in der Praxis demonstrieren soll:

```
// string1.cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
   // Einen leeren String erzeugen - Default-Konstruktor
   string str1;
   // Einen String mit einem C-String erzeugen
   string str2("Ein String aus einem C-String");
   // Ein C-String
   char name[] = "Wolf Jürgen";
   // Die ersten vier Zeichen des C-Strings kopieren
   string str3( name, 4 );
   // Einen String aus einem bereits
   // existierenden String erzeugen
   string str4( str3 );
   // Selbiges, nur werden jetzt Teilstrings übernommen
   string str5( str2, 0, 3 );
   string str6( str2, 4, 6 );
   // Einen String mit Zeichen füllen
   string str7( 20, '-' );

   // Die einzelnen Strings ausgeben
   cout << "str1: " << str1 << endl;
   cout << "str2: " << str2 << endl;
   cout << "str3: " << str3 << endl;
   cout << "str4: " << str4 << endl;
   cout << "str5: " << str5 << endl;
   cout << "str6: " << str6 << endl;
   cout << "str7: " << str7 << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
str1:
str2: Ein String aus einem C-String
str3: Wolf
str4: Wolf
str5: Ein
str6: String
str7: --------------------
```

In der String-Bibliothek sind sämtliche Operatoren global überladen. Daher ist es auch möglich, die Operatoren << bzw. >> zur Ein- bzw. Ausgabe zu verwenden.

7.1.4 Zuweisungen

Einem String können Sie auch einen anderen String per Zuweisungsoperator (=) zuweisen. Selbst wenn der Platz des Ziel-Strings nicht groß genug ist, wird der benötigte Platz neu reserviert, und nicht mehr verwendeter Platz wird freigegeben. Dabei ist der Zuweisungsoperator (`operator=()`) so überladen, dass auch ein C-String oder gar einzelne Zeichen an einen String zugewiesen werden können. Dabei sind auch Mehrfachzuweisungen erlaubt. Hierzu ein kurzes Listing, das die Möglichkeiten, den Zuweisungsoperator in Verbindung mit Strings zu verwenden, demonstriert:

```
// string2.cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
   // Leere Strings erzeugen
   string str1, str2, str3, str4, str5, str6;
   // Ein C-String
   char cstring[] =  "Bin ein C-String";

   cout << "Bitte den Namen eingeben: ";
   getline( cin ,str1 );
   // Einfache Zuweisung
   str2 = str1;
   // Einfache Zuweisung eines C-Strings
   str3 = cstring;
   // Eine Zuweisung eines einzelnen Zeichens
   str4 = 'A';
   // Mehrfachzuweisung
   str5 = str6 = "Eine Mehrfachzuweisung";

   // Die einzelnen Strings ausgeben
   cout << "str1: " << str1 << endl;
   cout << "str2: " << str2 << endl;
   cout << "str3: " << str3 << endl;
   cout << "str4: " << str4 << endl;
   cout << "str5: " << str5 << endl;
   cout << "str6: " << str6 << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Bitte den Namen eingeben: Jürgen Wolf
str1: Jürgen Wolf
str2: Jürgen Wolf
str3: Bin ein C-String
str4: A
str5: Eine Mehrfachzuweisung
str6: Eine Mehrfachzuweisung
```

Hinweis [«]

Auf die Methode `getline()`, die zum Einlesen von `string` verwendet wurde, wird in Kürze näher eingegangen.

Neben dem Weg über den Zuweisungsoperator gibt es noch die Methode `assign()` mit weiteren Möglichkeiten. Beispielsweise lässt sich folgendermaßen ein Teil-String zuweisen:

```
string str1;
string str2("Eierkopf");

// Zuweisung von str2 ab Pos.4, 4 Zeichen -> "kopf"
str1.assign( str2, 4, 4 );
cout << str1 << endl;   // kopf
```

Hierzu weitere `assign`-Methoden, die alle den Rückgabewert `*this` haben:

Methode	Beschreibung
`string& assign( const string& str );`	Ersetzt den aktuellen String (`*this`) durch den String `str`.
`string& assign( const string& str, size_type pos, size_type n );`	Ersetzt den aktuellen String (`*this`) durch den String `str` mit `n` Länge ab der Position `pos`. Dabei wird höchstens bis zum letzten Zeichen von `str` kopiert. Ist `*this` länger als `n`, wird `*this` auf `n` gekürzt.
`string& assign( const char* s, size_type n );`	Ersetzt den aktuellen String (`*this`) durch den C-String `s` mit maximal `n` Zeichen. Es wird höchstens bis zum letzten Zeichen von `s` kopiert.
`string& assign( const char* s );`	Ersetzt den aktuellen String (`*this`) durch den C-String `s`.
`string& assign( size_type n, char ch );`	Ersetzt den aktuellen String (`*this`) durch einen String mit der Länge `n`, die mit dem Zeichen `ch` aufgefüllt werden.

Tabelle 7.4 Zuweisungsmethoden

7.1.5 Elementzugriff

Der Zugriff auf einzelne Zeichen erfolgt auch bei der String-Bibliothek wie üblich mit Hilfe eines Indexoperators `[]`. Soll im Programm außerdem noch eine Bereichsüberprüfung stattfinden, kann statt des `[]`-Operators auch die Methode `at()` verwendet werden (siehe Tabelle 7.5). Diese löst im Fehlerfall eine Exception vom Typ `out_of_range` aus (siehe Abschnitt 6.5.2, »Anwenden der Standard-Exceptions«).

Methode	Beschreibung
`const char operator[] ( size_type pos ) const;`	Gibt das Element des Strings in der Position `pos` zurück. Ist `pos==size()`, wird das Stringende-Zeichen zurückgegeben. Wenn `pos>size()`, ist das Verhalten undefiniert.
`char& operator[](size_type pos);`	Dito, nur wird eine Referenz zurückgegeben.
`const char& at ( size_type pos ) const;`	Liefert eine konstante Referenz des Strings in der Position `pos` zurück. Falls `pos>=size()`, wird eine Exception vom Typ `out_of_range` ausgelöst.
`char& at ( size_type pos );`	Dito, nur ist die zurückgegebene Referenz nicht konstant.

Tabelle 7.5 Methoden für den Elementzugriff

Hierzu ein Beispiel, das den Zugriff auf die einzelnen Elemente eines Strings der `string`-Bibliothek demonstrieren soll:

```
// string3.cpp
#include <iostream>
#include <string>
#include <stdexcept>
using namespace std;

int main() {
   string str1 = "Tag und Nacht";
   const string str2("Ich bin konstant");
   // Einzelnes Zeichen einem char zuweisen
   char ch = str1[0];
   // Erstes Zeichen überschreiben
   str1[0] = 'S';
   // Zeichen mit Hilfe einer Referenz überschreiben
   char& chref = str1[8];
   chref = 'P';
```

```
   cout << "str1   : " << str1 << endl;
   cout << "ch     : " << ch << endl;
   // oder auch ...
   cout << "str1[5]: " << str1[5] << endl;
   // Hiermit wird die zweite Version des
   // Operators [] aufgerufen
   cout << "str2[1]: " << str2[1] << endl;
   // Dies ist allerdings mit einem konstanten
   // String nicht möglich
   // str2[0] = 'A';

   try {
      // Ok
      str1.at(0) = 'T';
      // Fehler - Bereichsüberschreitung
      cout << str1.at(20);
   }
   catch( out_of_range& ex ) {
      cout << "Exception (out_of_range): ";
      cout << ex.what() << endl;
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
str1   : Sag und Pacht
ch     : T
str1[5]: n
str2[1]: c
Exception (out_of_range): basic_string::at
```

7.1.6 Länge und Kapazität ermitteln bzw. ändern

Neben dem dynamischen Element sind in der Klasse `string` zwei weitere private Elemente deklariert – eins, das die String-Länge enthält, und eins, das die Kapazität des Strings enthält. Die String-Länge gibt die tatsächliche Anzahl von Zeichen an, die im String enthalten sind. Mit der Kapazität wird die maximale Anzahl von Zeichen angegeben, die in einem String gespeichert werden können, ohne dass erneut Speicher angefordert werden muss (Methoden dazu siehe Tabellen 7.6 und 7.7).

Methode	Beschreibung
`size_type size() const;`	aktuelle Anzahl der im String gespeicherten Zeichen
`size_type length() const;`	dito, wie `size()`
`bool empty() const;`	Wenn der String leer ist, wird `true` zurückgegeben.
`size_type max_size() const;`	Liefert die maximal mögliche Länge des Strings (`npos-1`) zurück.
`size_type capacity() const;`	Gibt die maximale Anzahl von Zeichen zurück, die im String gespeichert werden können, ohne dass erneut Speicher reserviert werden muss.

Tabelle 7.6 Bestimmung der Länge und Kapazität eines Strings

Methode	Beschreibung
`void resize( size_type n, char ch );`	Ändert die Länge des Strings wie folgt: Ist `n<=size()`, wird der String bis auf Länge `n` gekürzt. Ist `n>size()`, wird ab der Position `size()` bis `n-1` mit dem Zeichen `ch` gefüllt. Ist `n>max_size()`, wird eine Exception vom Typ `length_error` ausgelöst.
`void resize( size_type n );`	dito, nur ohne das Auffüllen mit Zeichen
`void clear( );`	Löscht alle Zeichen im String und setzt die Länge auf `0` zurück.
`void reserve( size_type n );`	Ändert die Kapazität des Strings auf `n`.

Tabelle 7.7 Änderung von Länge und Kapazität eines Strings

Hierzu ein Beispiel, das die Methoden zur Bestimmung und Änderung der Länge bzw. Kapazität in der Praxis demonstriert:

```
// string4.cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
   string str1;
   string str2("VBIQVE DAEMON ... VBIQVE DEVS ...");
   if( str1.empty() == true ) {
      cout << "str1 ist leer" << endl;
      cout << "str1: Max. Kapazität: "
           << str1.capacity() << endl;
      // Kapazität ändern
      str1.reserve( 10 );
      cout << "str1: Max. Kapazität (nach reserve): "
           << str1.capacity() << endl;
   }
```

```
   else {
      cout << "str1 ist nicht leer" << endl;
   }
   if( str2.empty() == true ) {
      cout << "str2 ist leer" << endl;
   }
   else {
      cout << "str2 ist nicht leer" << endl;
      cout << "str2: " << str2 << "(== "
           << str2.size() << " Zeichen)" << endl;
      cout << "str2: Max. Kapazität: "
           << str2.capacity() << endl;
   }
   // str2 verkleinern
   str2.resize( 14 );
   cout << "str2 (nach resize): " << str2 << "(== "
        << str2.size() << " Zeichen)" << endl;
   cout << "str2: Max. Kapazität: " << str2.capacity()
        << endl;

   // str1 vergrößern und mit Zeichen befüllen
   str1.resize( 20, 'x' );
   cout << "str1 (nach resize): " << str1 << "(== "
        << str1.size() << " Zeichen)" << endl;

   // Nochmals, nur ist der String jetzt nicht leer
   str2.resize( 17, '6' );
   cout << "str2 (nach resize): " << str2 << "(== "
        << str2.size() << " Zeichen)" << endl;

   // Zeichen in str1 löschen und auf 0 setzen
   str1.clear();
   if( str1.empty() == true ) {
      cout << "str1 ist jetzt leer (nach clear)" << endl;
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
str1 ist leer
str1: Max. Kapazität: 0
str1: Max. Kapazität (nach reserve): 10
str2 ist nicht leer
str2: VBIQVE DAEMON ... VBIQVE DEVS ...(== 33 Zeichen)
```

```
str2: Max. Kapazität: 33
str2 (nach resize): VBIQVE DAEMON (== 14 Zeichen)
str2: Max. Kapazität: 33
str1 (nach resize): xxxxxxxxxxxxxxxxxxx(== 20 Zeichen)
str2 (nach resize): VBIQVE DAEMON 666(== 17 Zeichen)
str1 ist jetzt leer (nach clear)
```

7.1.7 Konvertieren in einen C-String

Die Konvertierung eines C-Strings in einen String der Klasse `string` erfolgt stets implizit vom Konvertierungskonstruktor. Die Umwandlung eines Strings der Klasse `string` in einen C-String lässt sich allerdings nicht mehr implizit durchführen. Das muss so sein, da es sonst bei gemischten Ausdrücken zu Mehrdeutigkeiten kommen könnte. Für solche Zwecke stehen Ihnen weitere Methoden der String-Bibliothek zur Verfügung:

Methode	Beschreibung
`size_type copy( char *s, size_type n, size_type pos = 0) const;`	Kopiert `n` Zeichen vom aktuellen String (`*this`) ab der Position `pos` in ein durch `s` adressiertes `char`-Array. Das Stringende-Zeichen wird nicht kopiert. Der Rückgabewert ist die Anzahl der kopierten Zeichen.
`const char* c_str() const;`	Gibt die Zeichen des aktuellen Strings (`*this`) als C-String mit der Länge `size()`+1 zurück (Read-only).
`const char* data() const;`	Gibt die Anfangsadresse des gespeicherten Strings zurück (Read-only).

Tabelle 7.8 Konvertieren von Strings

Hierzu ein Beispiel, das alle drei Methoden zur Umwandlung eines Strings in einen C-String zeigen soll:

```
// string5.cpp
#include <iostream>
#include <string>
#include <cstring>
using namespace std;

int main() {
   char buffer[20];
   char buffer2[20];
   char buffer3[20];
   const char *ptr;
   string str("Yin-Yang");
   int n;
```

```
    // String in C-String mit c_str kopieren
    strncpy( buffer, str.c_str(), sizeof(buffer));
    cout << "str    : " << str << endl;
    cout << "buffer : " << buffer << endl;
    ptr = str.data();
    cout << "ptr    : " << ptr << endl;

    // 3 Zeichen nach buffer2 kopieren
    n = str.copy( buffer2, 3, 0 );
    buffer2[n] = '\0';
    cout << "buffer2: " << buffer2 << endl;

    // 4 Zeichen ab Pos. 4 nach buffer3 kopieren
    n = str.copy( buffer3, 4, 4 );
    buffer3[n] = '\0';
    cout << "buffer3: " << buffer3 << endl;
    return 0;
}
```

Das Programm bei der Ausführung:

```
str    : Yin-Yang
buffer : Yin-Yang
ptr    : Yin-Yang
buffer2: Yin
buffer3: Yang
```

7.1.8 Manipulation von Strings

Sozusagen das Sahnehäubchen und ein weiterer Grund, die Bibliothek `string` zu verwenden, sind die zahlreichen Methoden zur Manipulation von Strings. Dabei sind Operationen für das Einfügen, Anhängen, Ersetzen und Löschen von Strings bzw. Teil-Strings vorhanden. Alle Methoden sind mehrfach überladen, so dass diese jederzeit sehr vielseitig anwendbar sind. Das geht so weit, dass C-Strings auch als Argumente fungieren können. Hier zunächst ein Überblick über die Methoden, mit denen Sie Strings manipulieren können:

Methode	Beschreibung
`string& insert( size_type pos, const string& str );`	Fügt den String `str` bei der Position `pos` des aktuellen Strings (`*this`) ein. Die Zeichen hinter `str` bleiben erhalten und werden »nach hinten geschoben«. Der Speicherplatz wird dabei automatisch reserviert (erweitert).

Tabelle 7.9 Manipulation von Strings

Methode	Beschreibung
`string& insert( size_type pos, const char* s );`	Dito, nur wird statt eines Strings der Klasse `string` ein C-String verwendet.
`string& insert( size_type pos1, const string& str, size_type pos2, size_type n );`	Fügt in Position `pos1` des aktuellen Strings (`*this`) `n` Zeichen aus dem String `str` beginnend ab der Position `pos2` ein. Dabei wird höchstens bis zum letzten Zeichen von `str` kopiert.
`string& insert( size_type pos, const char* s, size_type n );`	Fügt in der Position `pos` des aktuellen Strings (`*this`) `n` Zeichen des C-Strings `s` ein. Es werden allerdings höchsten `strlen(s)` Zeichen eingefügt.
`string& insert( size_type pos, size_type n, char c );`	Fügt in der Position `pos` des aktuellen Strings (`*this`) `n`-mal das Zeichen `c` ein.
`string& append( const string& str );`	Hängt den String `str` an den aktuellen String (`*this`) an.
`string& append( const char* s );`	Hängt den C-String `s` an den aktuellen String (`*this`) an.
`string& append( const string& str, size_type pos, size_type n );`	Hängt `n` Zeichen des Strings `str` beginnend mit der Position `pos` an den aktuellen String (`*this`) an.
`string& append( const char* s, size_type n );`	Hängt `n` Zeichen des C-Strings `s` an den aktuellen String (`*this`) an. Dabei werden höchstens `strlen(s)` Zeichen angehängt.
`string& append( size_type n, char ch );`	Hängt `n`-mal das Zeichen `ch` an das Ende des aktuellen Strings (`*this`).
`string& erase( size_type pos = 0, size type n = npos );`	Löscht ab der Position `pos` `n` Zeichen aus dem aktuellen String (`*this`). Hierbei wird höchstens bis zum letzten Zeichen von `*this` gelöscht. Wird das zweite Argument weggelassen, wird automatisch von `pos` bis zum Ende des Strings gelöscht. Ohne Argumente wird der komplette String gelöscht.
`string& replace( size_type pos, size_type n, const string& str );`	Ersetzt im aktuellen String (`*this`) `n` Zeichen ab der Position `pos` durch den String `str`. Dabei kann höchstens bis zum Ende von `*this` ersetzt werden.

Tabelle 7.9 Manipulation von Strings (Forts.)

Methode	Beschreibung
`string& replace( size_type pos1, size_type n1, const string& str, size_type pos2, size_type n2 );`	Ersetzt `n1` Elemente ab der Position `pos1` des aktuellen Strings (`*this`) durch `n2` Elemente ab der Position `pos2` des Strings `str`. Hierbei wird auch nur höchstens bis zum Ende von `*this` bzw. `str` ersetzt.
`string& replace( size_type pos, size_type n, const char* s );`	Ersetzt ab der Position `pos` im aktuellen String (`*this`) `n` Zeichen durch den C-String `s`. Es wird allerdings höchstens bis zum letzten Zeichen von `*this` ersetzt.
`string& replace( size_type pos, size_type n1, const char* s, size_type n2 );`	Ersetzt `n1` Elemente ab der Position `pos` des aktuellen Strings (`*this`) durch `n2` Elemente des C-Strings `s`. Hierbei wird höchstens bis zum Ende von `*this` bzw. `s` ersetzt.
`void swap( string& str );`	Vertauscht den Inhalt des aktuellen Strings (`*this`) durch den Inhalt von `str`.
`string substr( size_type pos = 0, size_type n = npos );`	Erzeugt aus dem aktuellen String (`*this`), angefangen bei der Position `pos` bis zur Position `n`, einen Teil-String. Der neue String wird zurückgegeben.

Tabelle 7.9 Manipulation von Strings (Forts.)

Hierzu ein Listing, das einige dieser String-Manipulationsmethoden in der Praxis demonstriert:

```
// string6.cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
   string str1("Die Vögel singen ...");
   string str2(" nicht mehr");
   string str3("...trillili ...trillilu");
   string str4("Menschen");
   string str5("Ist die Welt eine Scheibe?");
   char cstr1[] = " immer";

   // String an Pos. 16 einfügen
   str1.insert(16 , str2 );
   cout << "str1 (insert) : " << str1 << endl;
   // C-String an Pos 22. einfügen
   str1.insert( 22, cstr1 );
   cout << "str1 (insert) : " << str1 << endl;
```

```
    // Von str3 ab Pos. 3 9 Zeichen in str1
    // ab Pos. 10 kopieren
    str1.insert( 10, str3, 3, 9 );
    cout << "str1 (insert) : " << str1 << endl;

    // Hängt str3 ab Pos. 15 8 Zeichen am Ende von str1
    str1.append( str3, 15, 8 );
    cout << "str1 (append) : " << str1 << endl;

    // Ab Pos. 10 9 Zeichen entfernen
    str1.erase( 10, 9 );
    cout << "str1 (erase)  : " << str1 << endl;
    // Ab Position 22 bis zum Ende Zeichen löschen
    str1.erase( 22 );
    cout << "str1 (erase)  : " << str1 << endl;

    // Ab Pos. 4 5 Zeichen durch den String str4 ersetzen
    str1.replace( 4, 5, str4 );
    cout << "str1 (replace): " << str1 << endl;
    // Ab der Position 4 bis zum Ende den
    // String str5 ab der Pos. 8 bis zum Ende ersetzen
    str1.replace( 4, str1.size(), str5, 8, str5.size() );
    cout << "str1 (replace): " << str1 << endl;

    // Teil-String aus str1 ab Pos. 4,4 Zeichen extrahieren
    string str6 = str1.substr( 4, 4 );
    cout << "str6 (substr) : " << str6 << endl;
    return 0;
}
```

Das Programm bei der Ausführung:

```
str1 (insert):Die Vögel singen nicht mehr ...
str1 (insert):Die Vögel singen nicht immer mehr ...
str1 (insert):Die Vögel trillili singen nicht immer mehr ...
str1 (append):Die Vögel trillili singen nicht immer mehr ...trillilu
str1 (erase) : Die Vögel singen nicht immer mehr ...trillilu
str1 (erase) : Die Vögel singen nicht
str1 (replace): Die Menschen singen nicht
str1 (replace): Die Welt eine Scheibe?
str6 (substr) : Welt
```

7.1.9 Suchen in Strings

Für das Suchen in einem String von Teil-Strings oder gar einzelnen Zeichen stehen Ihnen die Methoden `find()` und `rfind()` zur Verfügung. Der Unterschied

zwischen den beiden Methoden liegt in der Suchrichtung. `find()` sucht standardmäßig vom Anfang des Strings (oder einfach von links nach rechts) nach dem Teil-String. `rfind()` hingegen sucht von hinten bzw. von rechts nach links nach dem ersten Vorkommen des Teil-Strings (siehe auch Tabelle 7.10). Für die Suche nach einem Zeichen aus einer Menge von Zeichen gibt es die Methoden `find_first_of()` und `find_last_of()`. Wenn dagegen ein Zeichen gesucht werden soll, das nicht in der vorgegebenen Menge enthalten ist, stehen dafür die Methoden `find_first_not_of()` und `find_last_not_of()` zur Verfügung (siehe Tabellen 7.11 und 7.12).

Methode	Beschreibung
`size_type find ( const string& str, size_type pos = 0 ) const;`	Sucht im aktuellen String (`*this`) ab der Position `pos` nach dem ersten Vorkommen von `str`.
`size_type find ( const char* s, size_type pos = 0 ) const;`	Sucht im aktuellen String (`*this`) ab der Position `pos` nach dem ersten Vorkommen des C-Strings `s`.
`size_type find ( const char* s, size_type pos, size_type n ) const;`	Sucht im aktuellen String (`*this`) ab der Position `pos` nach dem ersten Vorkommen des Teil-Strings, der aus den ersten `n` Zeichen des C-Strings `s` besteht.
`size_type find ( char ch, size_type pos = 0 ) const;`	Sucht im aktuellen String (`*this`) ab der Position `pos` nach dem ersten Vorkommen des Zeichens `ch`.
`size_type rfind ( const string& str, size_type pos = 0 ) const;`	Sucht im aktuellen String (`*this`) ab der Position `pos` nach dem letzten Vorkommen von `str`.
`size_type rfind ( const char* s, size_type pos = 0 ) const;`	Sucht im aktuellen String (`*this`) ab der Position `pos` nach dem letzten Vorkommen des C-Strings `s`.
`size_type rfind ( const char* s, size_type pos, size_type n ) const;`	Sucht im aktuellen String (`*this`) ab der Position `pos` nach dem letzten Vorkommen des Teil-Strings, der aus den ersten `n` Zeichen des C-Strings `s` besteht.
`size_type rfind ( char ch, size_type pos = 0 ) const;`	Sucht im aktuellen String (`*this`) ab der Position `pos` nach dem letzten Vorkommen des Zeichens `ch`.

Tabelle 7.10 Suche nach Zeichenfolgen

Hinweis [«]

Alle `find()`- und `rfind()`-Methoden geben die gefundene Position zurück oder `npos`, wenn nichts gefunden wurde.

Hierzu ein einfaches Beispiel, das `find()` und `rfind()` in der Praxis demonstriert:

```
// string7.cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
   string str("Es gruent so gruen ...!");
   // Nach einem C-String suchen
   cout << "Erstes Auftreten  (gruen): "
        << str.find("gruen")  << endl;
   cout << "Letztes Auftreten (gruen): "
        << str.rfind("gruen") << endl;
   // Nach einzelnen Zeichen suchen
   cout << "Erstes Auftreten  ('s'): "
        << str.find('s')  << endl;
   cout << "Letztes Auftreten ('s'): "
        << str.rfind('s') << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Erstes Auftreten  (gruen): 3
Letztes Auftreten (gruen): 13
Erstes Auftreten  ('s'): 1
Letztes Auftreten ('s'): 10
```

Natürlich lässt sich hiermit auch ohne Probleme das »Suchen und Ersetzen« implementieren. Hierzu benötigen Sie lediglich die Methoden `find()` und `replace()`:

```
// replString.cpp
#include <iostream>
#include <string>
#include <cstring>
using namespace std;

void find_replace( char* s1, char* s2 );

int main( int argc, char **argv ) {
   if( argc != 3 ) {
      cout << "Verwendung: replString oldString newString "
           << "[ < text.txt ] [ > neu_text.txt ]" << endl;
      return 1;
   }
```

```
   find_replace( argv[1], argv[2] );
   return 0;
}

void find_replace( char* s1, char* s2 ) {
   string::size_type pos, anzahl = strlen(s1);
   string line;
   line.reserve(256);
   while( getline( cin, line ) ) {
      pos = 0;
      while((pos = line.find( s1, pos ))!=string::npos ) {
         line.replace( pos, anzahl, s2 );
         pos++;
      }
      cout << line << endl;
   }
}
```

Zur Ausführung des Beispiels sollten Sie eine Kommandozeile verwenden:

```
$ replString Meier Wolf
Mein Name ist Meier
Meine Name ist Wolf
Nein, Meier meine ich
Nein, Wolf meine ich
```

Natürlich können Sie die Standardeingabe auch folgendermaßen umleiten:

```
$ replString Meier Wolf < datei.txt
```

Dabei werden in der Datei *datei.txt* alle Teil-Strings `Meier` als `Wolf` auf die Standardausgabe ausgegeben. Auch die Ausgabe können Sie natürlich in eine Datei umleiten:

```
$ replString Meier Wolf < datei.txt > neue_Datei.txt
```

Weiter geht es mit den Methoden, mit denen Sie nach Zeichen aus einer Menge von Zeichen suchen können:

Methode	Beschreibung
`size_type find_first_of( const string& str, size_type pos = 0 ) const;`	Sucht im aktuellen String (`*this`), angefangen bei der Position `pos`, nach dem ersten Vorkommen eines der im String `str` enthaltenen Zeichen.
`size_type find_first_of( const char* s, size_type pos = 0 ) const;`	Sucht im aktuellen String (`*this`), angefangen bei der Position `pos`, nach dem ersten Vorkommen eines der im C-String `s` enthaltenen Zeichen.

Tabelle 7.11 Suche nach Zeichen aus einer Menge von Zeichen

Methode	Beschreibung
`size_type find_first_of( const char* s, size_type pos, size_type n ) const;`	Sucht im aktuellen String (`*this`), angefangen bei der Position `pos`, nach dem ersten Vorkommen eines der ersten `n` Zeichen des C-Strings `s`.
`size_type find_first_of( char ch, size_type pos = 0 ) const;`	Sucht im aktuellen String (`*this`), angefangen bei der Position `pos`, nach dem ersten Vorkommen des Zeichens `ch`.
`size_type find_last_of( const string& str, size_type pos = 0 ) const;`	Sucht im aktuellen String (`*this`), angefangen bei der Position `pos`, nach dem letzten Vorkommen eines der im String `str` enthaltenen Zeichen.
`size_type find_last_of( const char* s, size_type pos = 0 ) const;`	Sucht im aktuellen String (`*this`), angefangen bei der Position `pos`, nach dem letzten Vorkommen eines der im C-String `s` enthaltenen Zeichen.
`size_type find_last_of( const char* s, size_type pos, size_type n ) const;`	Sucht im aktuellen String (`*this`), angefangen bei der Position `pos`, nach dem letzten Vorkommen eines der ersten `n` Zeichen des C-Strings `s`.
`size_type find_last_of( char ch, size_type pos = 0 ) const;`	Sucht im aktuellen String (`*this`), angefangen bei der Position `pos`, nach dem letzten Vorkommen des Zeichens `ch`.

Tabelle 7.11 Suche nach Zeichen aus einer Menge von Zeichen (Forts.)

[»]

Hinweis

Alle Methoden geben die gefundene Position zurück oder `npos`, wenn nichts gefunden wurde.

Als Gegenstück zu den eben gezeigten Methoden (siehe Tabelle 7.11) gibt es noch die `_not_`-Versionen, die nach einem Zeichen suchen, das in einer vorgegebenen Menge nicht enthalten ist. Auch hierzu ein Überblick über die Methoden:

Methode	Beschreibung
`size_type find_first_not_of( const string& str, size_type pos = 0 ) const;`	Sucht im aktuellen String (`*this`), angefangen bei der Position `pos`, nach dem ersten Vorkommen eines Zeichens, das nicht im String `str` enthalten ist.
`size_type find_first_not_of( const char* s, size_type pos = 0 ) const;`	Sucht im aktuellen String (`*this`), angefangen bei der Position `pos`, nach dem ersten Vorkommen eines Zeichens, das nicht im C-String `s` enthalten ist.
`size_type find_first_not_of( const char* s, size_type pos, size_type n ) const;`	Sucht im aktuellen String (`*this`), angefangen bei der Position `pos`, nach dem ersten Vorkommen eines Zeichens, das nicht in den ersten `n` Zeichen im C-String `s` enthalten ist.

Tabelle 7.12 Nach Zeichen suchen, die in einer vorgegebenen Menge nicht enthalten sind

Methode	Beschreibung
`size_type find_first_not_of( char ch, size_type pos = 0 ) const;`	Sucht im aktuellen String (`*this`), angefangen bei der Position `pos`, nach dem ersten Vorkommen eines Zeichens, das nicht dem Zeichen `ch` entspricht.
`size_type find_last_not_of( const string& str, size_type pos = 0 ) const;`	Sucht im aktuellen String (`*this`), angefangen bei der Position `pos`, nach dem letzten Vorkommen eines Zeichens, das nicht im String `str` enthalten ist.
`size_type find_last_not_of( const char* s, size_type pos = 0 ) const;`	Sucht im aktuellen String (`*this`), angefangen bei der Position `pos`, nach dem letzten Vorkommen eines Zeichens, das nicht im C-String `s` enthalten ist.
`size_type find_last_not_of( const char* s, size_type pos, size_type n ) const;`	Sucht im aktuellen String (`*this`), angefangen bei der Position `pos`, nach dem letzten Vorkommen eines Zeichens, das nicht in den ersten `n` Zeichen im C-String `s` enthalten ist.
`size_type find_last_not_of( char ch, size_type pos = 0 ) const;`	Sucht im aktuellen String (`*this`), angefangen bei der Position `pos`, nach dem letzten Vorkommen eines Zeichens, das nicht dem Zeichen `ch` entspricht.

Tabelle 7.12 Nach Zeichen suchen, die in einer vorgegebenen Menge nicht enthalten sind (Forts.)

Hierzu ein häufig zitiertes Beispiel, das einen String in einzelne Wörter zerlegt. Dabei sollen die einzelnen Wörter in einem Vektor (`vector`) gespeichert werden:

```
// string8.cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;

unsigned int splitString(
   const string& str, vector<string>& vals,
   const char *sep, unsigned int max_tokens );

int main( int argc, char **argv ) {
  vector<string> woerter;
  string str("Ich bin ein String mit mehreren Worten.");
  unsigned int count;
  count =  splitString( str, woerter, " \n\t", 0 );
  if( !count ) {
     cout << "Der String war vermutlich leer!?!" << endl;
     return 1;
  }
  cout << "Der String wurde in " << count
       << " Teile zerlegt" << endl;
```

```
  for( vector<string>::iterator iter = woerter.begin();
       iter != woerter.end(); iter++ ) {
     cout << *iter << endl;
   }
   return 0;
}

// Argumente:
// str        : String, der zerlegt werden soll
// vals       : Vektor, wo die Teilstrings gespeichert werden
// sep        : Trennzeichen, durch die zerlegt wird
// max_tokens : Max. Teilstrings - 0 == beliebig
unsigned int splitString(
   const string& str, vector<string>& vals,
   const char *sep, unsigned int max_tokens ) {
   // Ist der String leer?
   if (!str.length())
      return 0;

   // Führende Whitespaces, Newlines und Tabs entfernen
   string::size_type start = str.find_first_not_of(" \n\t");
   if (start == string::npos)
      return 0; // Nichts zu machen ...

   string::size_type stop;
   unsigned int tokens = 0;

   for ( unsigned int i = 0;
         (max_tokens == 0) || (i < max_tokens); ++i ) {
      if (max_tokens && ((i + 1) == max_tokens)) {
         stop = str.size();
      }
      else {
         stop = str.find_first_of(sep, start);
      }
      if (stop != string::npos) {
         // Sicher gehen, ob ein Token vorhanden ist
         if (stop > start) {
            vals.push_back(str.substr(start, stop - start));
            ++tokens;
         }
      }
      else if (start < str.size()) {
        vals.push_back(str.substr(start, str.size()-start));
        ++tokens;
```

```
        break;
      }
      else {
         break;
      }
      start = str.find_first_not_of(sep, stop);
   }
   return tokens;
}
```

Das Programm bei der Ausführung:

```
Der String wurde in 7 Teile zerlegt
Ich
bin
ein
String
mit
mehreren
Worten.
```

7.1.10 Strings vergleichen

Methode	Beschreibung
`int compare( const string& str ) const;`	Vergleicht den aktuellen String (`*this`) lexikalisch mit `str`. Ist das erste unterschiedliche Zeichen von `*this` kleiner als das von `str` oder ist `*this.size() <str.size()`, wird `<0` zurückgegeben. Sind beide Strings identisch, wird `0` zurückgegeben. Wenn im aktuellen String (`*this`) ein Zeichen größer ist das als von `str` oder `*this.size()>str.size()`, wird `>0` zurückgegeben.
`int compare( size_type pos, size_type n, const string& str ) const;`	Dito, nur werden `n` Zeichen des aktuellen Strings (`*this`), angefangen bei der Position `pos`, mit dem String `str` verglichen.
`int compare( size_type n1, const string& str, size_type pos2, size_type n2 ) const;`	Dito, nur werden `n1` Zeichen des aktuellen Strings (`*this`) mit `n2` Zeichen des Strings `str`, angefangen bei der Position `pos`, verglichen.
`int compare( const char *s ) const;`	Dito, nur wird hierbei der aktuelle String mit dem C-String `s` verglichen.

Tabelle 7.13 Vergleichsmethoden von Strings

[»] **Hinweis**

Die Methode `compare()` ist so etwas wie das Gegenstück zu `str(n)cmp()` aus der C-Standardbibliothek.

Hierzu ein Beispiel, wie sich die Methode `compare()` in der Praxis auf Strings anwenden lässt:

```
// string9.cpp
#include <iostream>
#include <string>
using namespace std;

int main( int argc, char **argv ) {
   string s1("aaaa");
   string s2("aaab");
   char cs[] = "aaaa";
   int ret;
   ret = s1.compare(s2);
   if( ret < 0 ) {
      cout << "s1 ist lexikografisch kleiner als s2"
           << endl;
   }
   else if( ret > 0 ) {
      cout << "s1 ist lexikografisch größer als s2" << endl;
   }
   else { // dann eben gleich ...
      cout << "s1 und s2 sind identisch" << endl;
   }
   ret = s1.compare( 0, 2, s2, 0, 2 );
   cout << "Vergleicht die ersten beiden Zeichen ..."
        << endl;
   if( ret < 0 ) {
      cout << "s1 ist lexikografisch kleiner als s2"
           << endl;
   }
   else if( ret > 0 ) {
      cout << "s1 ist lexikografisch größer als s2" << endl;
   }
   else { // dann eben gleich ...
      cout << "s1 und s2 sind identisch" << endl;
   }
   ret = s1.compare(cs);
   if( ret < 0 ) {
      cout << "s1 ist lexikografisch kleiner als cs"
```

```
            << endl;
   }
   else if( ret > 0 ) {
      cout << "s1 ist lexikografisch größer als cs" << endl;
   }
   else { // dann eben gleich ...
      cout << "s1 und cs sind identisch" << endl;
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
s1 ist lexikografisch kleiner als s2
Vergleicht die ersten beiden Zeichen ...
s1 und s2 sind identisch
s1 und cs sind identisch
```

7.1.11 Die (überladenen) Operatoren

Die String-Bibliothek bietet eine Menge globaler überladener Operatoren an, mit denen sich recht komfortabel eine Menge weiterer Operationen durchführen lassen.

Operator	Beschreibung
`string& operator+=( Param );`	Als Parameter kommen hier `string`, `char*` und `char` in Frage. Mit diesem Operator hängen Sie praktisch einen String oder C-String bzw. ein einzelnes Zeichen an den aktuellen String (`*this`). Diese Methode ist gleichwertig zur Methode `append()`.
`string& operator+( Param );`	Der +-Operator lässt sich ebenfalls in mehreren Mixturen verwenden. Auch hiermit hängen Sie einen String (`string`), ein Zeichen (`char`) oder einen C-String (`char*`) an einen anderen String, ein Zeichen oder einen C-String. Zurückgegeben wird der neue String.
`istream& operator>>( istream& is, string& str );`	Damit wird ein Zeichen vom Eingabestream eingelesen und an den String `str` gehängt. Zwischenraumzeichen oder das Zeilenende werden nicht eingelesen und beenden den Lesevorgang. Es werden maximal `str.max_size()` Zeichen eingelesen oder, wenn eine Feldbreite gesetzt wurde, `is.width()` Zeichen.
`ostream& operator<<( ostream& os, string& str );`	Sendet den String `str` an den Ausgabestream.

Tabelle 7.14 Überladene Operatoren der String-Bibliothek

Operator	Beschreibung
==	Vergleicht zwei Strings auf Gleichheit. Gibt `true` zurück, wenn gleich, ansonsten `false`. `str1==str2` ist äquivalent zu `str1.compare(str2)==0`.
!=	Vergleicht zwei Strings auf Ungleichheit. Gibt `true` zurück, wenn die Strings ungleich sind, ansonsten `false`. `str1!=str2` ist äquivalent zu `!(str1==str2)`.
<	Überprüft, ob der linke String lexikografisch kleiner ist als der rechte. Gibt `true` zurück, wenn das zutrifft, ansonsten `false`. `str1 < str2` ist äquivalent zu `str1.compare(str2)<0`.
<=	Überprüft, ob der linke String lexikografisch kleiner als oder gleich groß wie der rechte ist. Gibt `true` zurück, wenn das zutrifft, ansonsten `false`. `str1 <= str2` ist äquivalent zu `str1.compare(str2)<=0`.
>	Überprüft, ob der linke String lexikografisch größer ist als der rechte. Gibt `true` zurück, wenn das zutrifft, ansonsten `false`. `str1 > str2` ist äquivalent zu `str1.compare(str2)>0`.
>=	Überprüft, ob der linke String lexikografisch größer als oder gleich groß wie der rechte ist. Gibt `true` zurück, wenn das zutrifft, ansonsten `false`. `str1 >= str2` ist äquivalent zu `str1.compare(str2)>=0`.

Tabelle 7.15 Vergleichsoperatoren

Hierzu ein Beispiel, das die Verwendung der globalen überladenen Operatoren aus den Tabellen 7.14 und 7.15 in der Praxis zeigen soll:

```
// string10.cpp
#include <iostream>
#include <string>
using namespace std;

int main( int argc, char **argv ) {
   string s1("Dalai");
   string s2("Lama");
   char cs[] = "Seine Heiligkeit";
   // Ein einfaches char-Zeichen anhängen
   s1+=' ';
   // Einen String anhängen
   s1+=s2;
   cout << s1 << endl;

   string s3 = s1 + " - " + cs;
   cout << s3 << endl;

   string s4;
   cout << "String eingeben: ";
   cin >> s4;
   cout << "Ihre Eingabe " << s4 << endl;
```

```
   string s5("aaaa");
   string s6("aaab");

   if( s5 < s6) {
      cout << "s5 ist lexikografisch kleiner als s6"
           << endl;
   }
   else if( s5 > s6 ) {
      cout << "s5 ist lexikografisch größer als s6" << endl;
   }
   else { // dann eben gleich ... (s1 == s2)
      cout << "s5 und s6 sind identisch" << endl;
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Dalai Lama
Dalai Lama - Seine Heiligkeit
String eingeben: Test
Ihre Eingabe Test
s5 ist lexikografisch kleiner als s6
```

7.1.12 Einlesen einer ganzen Zeile

Zum Einlesen einer ganzen Textzeile ist der Shift-Operator >> nicht geeignet, da dieser nur die Zeichen bis zum nächsten Whitespace-Zeichen (Newline, Leerzeichen, Tabulator) einliest. Daher bietet auch die String-Bibliothek mit `getline()` eine Funktion an, mit der standardmäßig eine ganze Zeile, also bis zum nächsten Newline-Zeichen, eingelesen werden kann. Die Funktion ist überladen und in zwei Versionen vorhanden:

Funktion	Beschreibung
`istream& getline( istream& is, string& str, char delim );`	Liest eine komplette Zeile bis zum nächsten Newline-Zeichen oder bis zum mit `delim` angegebenen Zeichen vom Stream `is` ein und speichert die Eingabe im String `str`. Abgebrochen wird das Einlesen, wenn das Dateiendezeichen oder `str.max_size()`-Zeichen eingelesen wurde.
`istream& getline( istream& is, string& str );`	Dito, nur kann damit kein Extrazeichen angegeben werden, mit dem das Einlesen beendet wird.

Tabelle 7.16 Einlesen einer ganzen Textzeile

Auch hierzu ein Beispiel, das beide Möglichkeiten der Funktion `getline()` demonstrieren soll:

```
// string11.cpp
#include <iostream>
#include <string>
using namespace std;

int main( int argc, char **argv ) {
   string s1, s2;
   cout << "Bitte eine Zeile eingeben: ";
   getline( cin, s1 );
   cout << "Noch eine Zeile (aber ohne ein 'x'): ";
   getline( cin, s2, 'x' );
   cout << "Ihre 1.Eingabe: " << s1 << endl;
   cout << "Ihre 2.Eingabe: " << s2 << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Bitte eine Zeile eingeben: Ist nix drin
Noch eine Zeile (aber ohne ein 'x'): Ist nix drin
Ihre 1.Eingabe: Ist nix drin
Ihre 2.Eingabe: Ist ni
```

7.2 Ein-/Ausgabe Klassenhierarchie (I/O-Streams)

Das Ein-/Ausgabesystem von C++ mit ein paar Worten zu erklären dürfte wohl unmöglich sein. Aber wenn Sie das Prinzip der Klassen verstanden haben, dann wird Ihnen auch der Aufbau dieses Systems logisch erscheinen, denn es basiert komplett auf Klassen. Die I/O-Streams sind auch unter dem Namen *iostream-Bibliothek* bekannt. Der Hauptanwendungsbereich dieser Bibliothek liegt in der:

- Standardeingabe/-ausgabe
- Dateiverarbeitung
- String-Verarbeitung im Hauptspeicher

Ein Vorteil von Streams gegenüber den Ein-/Ausgabefunktionen von C sind die Typsicherheit – da der Compiler anhand des Argumenttyps selbst entscheidet, welche Ein-/Ausgaberoutine er aufruft – und die Erweiterbarkeit der Streams. So ist es ohne großen Aufwand möglich, durch Überladen der Operatoren << und >> selbstdefinierte Datentypen für die Ein-/Ausgabe-Streams zu verwenden.

Die iostream-Bibliothek ist hierarchisch organisiert, wobei die Klasse `ios` die Basisklasse der Familie darstellt (siehe Abbildung 7.2). In ihr sind alle Eigenschaften und Methoden definiert, die bei allen Streams gleich sind. Die einzelnen Templates sind mit einem Typ für Zeichen (in unserem Fall `char`) parametrisiert, so dass sich diese Klasse ohne großen Aufwand auch für erweiterte Zeichensätze verwenden lässt.

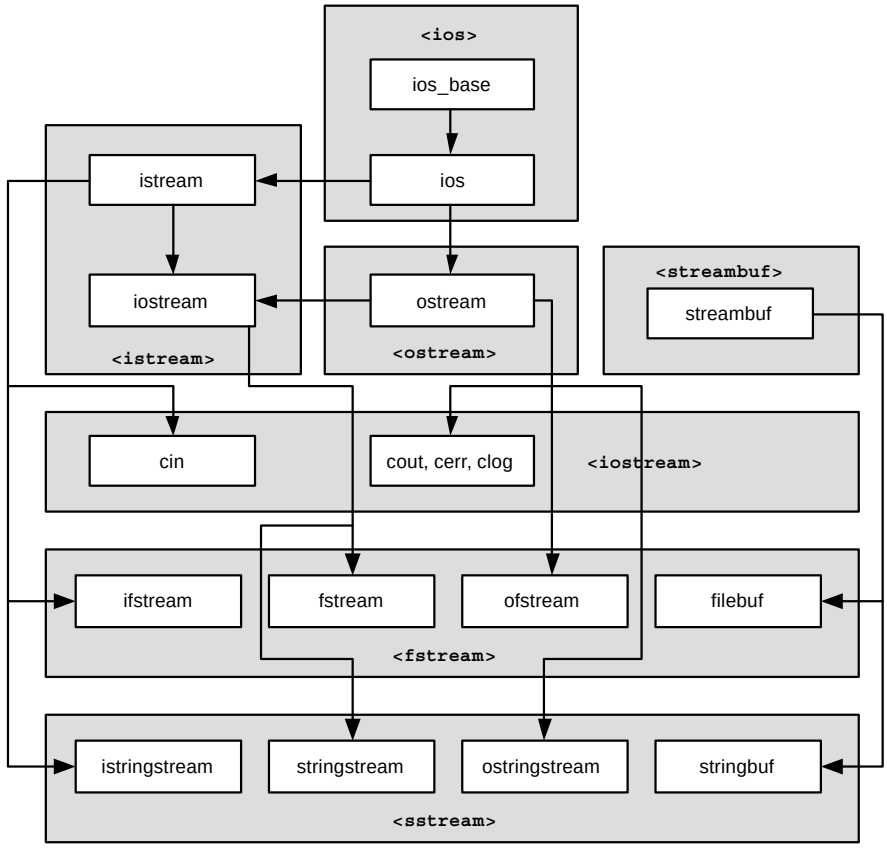

Abbildung 7.2 Klassen-Hierarchie von »iostream«

Die Basisklasse `ios` stellt somit die Grundlage für die Eigenschaften eines Streams zur Verfügung und erfüllt im Grunde folgende Aufgaben:

- Die Klasse `ios` stellt Eigenschaften und Methoden bereit, die die (formatierte) Ein-/Ausgabe kontrollieren.
- Die Klasse `ios` verwaltet auch den Puffer. Dazu enthält diese Klasse einen Zeiger auf ein Objekt der abgeleiteten Klasse `streambuf`, die den eigentlichen Datenstrom repräsentiert.

Die Klassen `istream` und `ostream` haben die Klasse `ios` als virtuelle Basisklasse. `istream` wird zum Einlesen und `ostream` zum Schreiben in Streams definiert. Dabei werden zum Beispiel die Operatoren << und >> für die elementaren Datentypen (auch Strings) überladen.

Die Klasse `ifstream` wird von der Klasse `istream` abgeleitet und ist eine Schnittstelle für das Lesen von Dateien. Das Gegenstück `ofstream`, das von `ostream` abgeleitet wurde, wird zum Schreiben in Dateien verwendet.

Die Klasse `iostream` bildet sich aus einer Mehrfachvererbung der Klassen `istream` und `ostream` und enthält Streams für die Ein-/Ausgabe. Die Klasse `fstream` erweitert diese Klasse um Funktionen für den Dateizugriff.

Alle Möglichkeiten zur formatierten Ein-/Ausgabe wie bei den Datei-Streams können auch für das Lesen und Schreiben von Strings verwendet werden. Hierfür stehen Ihnen die Klassen `istringstream`, `ostringstream` und `stringstream` zur Verfügung.

Die abstrakte Klasse `streambuf` ist eine gemeinsame Klasse für die Pufferklassen `filebuf` und `stringbuf`. `filebuf` ist der Puffer für die Dateieingabe/-ausgabe und `stringbuf` der Puffer für die String-Operationen im Hauptspeicher. Nicht in der Abbildung 7.2 vorhanden ist die Puffer-Klasse `stdiobuf`, die auf Basis der Standard-C-Bibliothek aufgebaut ist. Wichtig zu wissen ist, dass alle Klassen in der `ios`-Hierarchie mit einer von `streambuf` abgeleiteten Klasse für die Datenübertragung zusammenarbeiten.

Zum Abfangen einer Ausnahme (Exception) kann der Typ `ios::failure` von der Bibliothek `iostream` geworfen und abgefangen werden. Im Handle für eine abgefangene Exception können Sie auch hier die Methode `what()` aufrufen, die einen String mit der Fehlermeldung zurückgibt. Mehr zu den Exceptions finden Sie in Kapitel 6, »Exception-Handling«.

7.2.1 Klassen für Ein- und Ausgabe-Streams

Die von `ios` abgeleiteten Klassen `ostream` und `istream` stellen komfortable Methoden zur formatierten Ein-/Ausgabe zur Verfügung.

Die Klasse »ostream«

Die Klasse `ostream` wird für die Ausgabe in Streams verwendet. Als virtuelle Basisklasse besitzt sie `ios`. Da diese `public` vererbt wurde, stehen auch alle `public`-Elemente von `ios` zur Verfügung.

Eine besondere Fähigkeit von `ostream` ist, dass der Operator << für die formatierte Ausgabe der elementaren Datentypen, C-Strings, Manipulatoren und einige

Zeiger überladen ist (siehe Tabelle 7.17). Der Stream `cout` ist zum Beispiel ein Objekt der Klasse `ostream`. So wird zum Beispiel mit der Anweisung

```
cout << "Ein C-String";
```

die Methode `operator<<(const char*)` von `ostream` aufgerufen. Mit

```
cout << 'A';
```

würde die Methode `operator<<(char)` aufgerufen.

ostream& operator<<(TYP)	Beschreibung
`char`, `unsigned char`, `signed char`	Das entsprechende Zeichen wird ausgegeben.
`short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`	Wandelt die übergebene Ganzzahl für die Ausgabe in eine Zeichenfolge um. Die Zahlen werden standardmäßig dezimal ausgegeben.
`float`, `double`, `long double`	Wandelt die übergebene Gleitpunktzahl für die Ausgabe in eine Zeichenfolge um und gibt diese aus.
`const char*`, `const signed char*`, `const unsigned char*`	Gibt einen String ohne das Stringende-Zeichen aus.
`void*`	Gibt die Adresse in hexadezimaler Schreibweise aus.
`bool`	Gibt bei `true` `1` und bei `false` `0` aus.
`streambuf*`	Kopiert die Zeichen aus dem Puffer in den Ausgabepuffer des Streams, bis `EOF` oder ein Fehler auftritt.
`ios& (*pf)(ios&)`, `ostream& (*pf) (ostream&)`	Ruft den Manipulator von `ios` bzw. `ostream` auf, der mit `pf` adressiert wurde.

Tabelle 7.17 Operator << ist überladen für die formatierte Ausgabe.

Manipulatoren

In Tabelle 7.17 ist auch die Rede von Manipulatoren. Dabei ist jeweils eine Version für `ios` und eine für die Klasse `ostream` deklariert. Ein solcher Manipulator ist eine Schnittstelle, die dem Operator `<<` als Operand übergeben wird. Erhält der Operator `<<` einen solchen Operanden, ruft dieser eine entsprechende Funktion auf. In den Tabellen 7.18 bis 7.20 finden Sie einige Manipulatoren der Klasse `ostream` und `ios` aufgelistet, die Sie hierfür verwenden können.

Manipulator	Beschreibung
`flush`	Ausgabepuffer leeren
`endl`	einen Zeilenvorschub ausgeben und Ausgabepuffer leeren
`ends`	Stringende-Zeichen ausgeben und Ausgabepuffer löschen

Tabelle 7.18 Manipulatoren der Klasse »ostream«

Die Manipulatoren in den Tabellen 7.19 und 7.20 sind Manipulatoren der Klasse `ios` und stehen somit für die Ein- und Ausgabe-Streams zur Verfügung.

Manipulator	Beschreibung
`oct`	Das entsprechende Bit in `ios::basefield` wird gesetzt. Die nachfolgende Ein- bzw. Ausgabe erfolgt dann oktal.
`dec`	Das entsprechende Bit in `ios::basefield` wird gesetzt. Die nachfolgende Ein- bzw. Ausgabe erfolgt dann dezimal.
`hex`	Das entsprechende Bit in `ios::basefield` wird gesetzt. Die nachfolgende Ein- bzw. Ausgabe erfolgt dann hexadezimal.
`showbase`	Kennzeichnet bei der Ausgabe die Ganzzahl, beispielsweise bei oktalen Zahlen mit einer führenden `0` und bei einer hexadezimalen Zahl mit einem führenden `0x` bzw. `0X`. Dabei wird das Flag in `ios::showbase` gesetzt.
`noshowbase`	Löscht das Flag in `ios::showbase` – Gegenstück zu `showbase`.
`showpos`	Positive Werte werden mit dem Vorzeichen `+` ausgegeben. Setzt das Flag `ios::showpos`.
`noshowpos`	Löscht das Flag `ios::showpos` – Gegenstück zu `showpos`.
`uppercase`	Bei der Ausgabe von Zahlen werden Großbuchstaben verwendet, was im Grunde nur bei hexadezimalen Zahlen einen Sinn ergibt (`0X`). Setzt das Flag `ios::uppercase`.
`nouppercase`	Löscht das Flag `ios::uppercase` – Gegenstück zu `uppercase`.

Tabelle 7.19 Manipulatoren der Klasse »ostream«

Manipulator	Beschreibung
`fixed`	Darstellung als Festpunktzahl. Setzt das Flag `ios::fixed`.
`scientific`	Darstellung in exponentieller Schreibweise. Setzt das Flag `ios::scientific`.
`showpoint`	Bei der Ausgabe von Gleitpunktzahlen wird immer der Dezimalpunkt mit ausgegeben. Setzt das Flag `ios::showpoint`.
`noshowpoint`	Gegenstück zu `showpoint`. Löscht das Flag `ios::showpoint`.

Tabelle 7.20 Manipulatoren für Gleitpunktzahlen der Klasse »ios«

[»] Hinweis

Neben den Manipulatoren für Gleitpunktzahlen gibt es noch die Methode `precision()`, mit der Sie die Genauigkeit einstellen können. Ohne Argumente erhalten Sie die Anzahl der Genauigkeit (gewöhnlich 6), und mit einem ganzzahligen Wert als Argument können Sie diese Genauigkeit ändern.

Der Einsatz von Manipulatoren gestaltet sich, im Gegensatz zum Löschen bzw. Setzen von Flags über der Methode `setf()`, erheblich einfacher. Hierzu ein Beispiel, das zeigt, wie der Aufruf von Manipulatoren in der Praxis funktioniert:

```
// stream1.cpp
#include <iostream>
using namespace std;

int main( int argc, char **argv ) {
   int ival = 100;
   float fval = 100.111;
   char cs[] = "Ein C-String";

   // Gibt das Stringende-Zeichen aus
   cout << cs << ends << "|" << endl;
   // Ohne Stringende-Zeichen
   cout << cs << "|" << endl;

   cout << "Festpunktzahl  : " << fixed << fval << endl;
   cout << "Exponentiell   : " << scientific
        << fval << endl;
   cout << "Exponentiell   : " << uppercase
        << scientific << fval << endl;

   cout << "Oktal       : " << oct << ival << endl;
   cout << "Hexadezimal : " << hex << ival << endl;
   cout << "Dezimal     : " << dec << ival << endl;

   // Mit Basis ausgeben
   cout << "Mit Basis (showbase) ..." << showbase << endl;
   cout << "Oktal       : " << oct << ival << endl;
   cout << "Hexadezimal : " << hex << ival << endl;
   cout << "Dezimal     : " << dec << ival << endl;
   cout << noshowbase;

   // Positive Wert mit + anzeigen
   cout << "Mit Vorzeichen (showpos) ... "
        << showpos << endl;
   cout << "Dezimal     : " << dec << ival << endl;
   cout << noshowpos;

   // Mit Basis und Grossbuchstaben
   cout << "Mit Basis und Grossbuchstaben ... "
        << showbase << uppercase << endl;
   cout << "Hexadezimal : " << hex << ival << endl;
```

```
   cout << nouppercase;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Ein C-String |
Ein C-String|
Festpunktzahl  : 100.111000
Exponentiell   : 1.001110e+002
Exponentiell   : 1.001110E+002
Oktal       : 144
Hexadezimal : 64
Dezimal     : 100
Mit Basis (showbase) ...
Oktal       : 0144
Hexadezimal : 0X64
Dezimal     : 100
Mit Vorzeichen (showpos) ...
Dezimal     : +100
Mit Basis und Grossbuchstaben ...
Hexadezimal : 0X64
```

Eigene Manipulatoren

Die Operatoren >> (`istream`) und << (`ostream`) sind so überladen, dass diese auch Funktionen als Argumente zulassen. Gemäß Tabelle 7.17 (letzter Eintrag) muss der Prototyp einer solchen Funktion folgendermaßen aussehen:

```
// für istream und ostream
ios& funktion( ios& stream );

// für istream
istream& funktion ( istream& stream );
// für ostream
ostream& funktion ( ostream& stream );
```

Auch die Standardmanipulatoren besitzen eine solche Schnittstelle. Sie können den globalen Manipulator `endl` zum Beispiel folgendermaßen selbst definieren:

```
// stream2.cpp
#include <iostream>
using namespace std;

ostream& myendl( ostream& os ) {
   os << '\n';
   return os;
}
```

```
int main( int argc, char **argv ) {
   cout << "Ein Beispiel" << myendl;
   cout << "Die nächste Zeile " << myendl;
   return 0;
}
```

Hier wird der Operator `<<` für den Stream `cout` mit der Funktion `myendl()` als zweiter Operand aufgerufen. Die Implementierung des Operators `<<` wandelt diesen Aufruf in einen angepassten Funktionsaufruf wie folgt um:

```
myendl( cout );
```

So arbeiten im Grunde auch die Standardmanipulatoren. Wenn man sich nochmals die Syntax ansieht, sollte das einleuchten:

```
ostream& ostream::operator << (ostream&(*op) (ostream&))  {
    return (*op) (*this);
}
```

Es ist wichtig, dass die Manipulatorfunktion diese Schnittstelle besitzt. Wenn die Funktion einen anderen Typ besitzt, wird der Funktionsname bei der Übergabe an die Operatorfunktion in einen Zeiger vom Typ `void*` umgewandelt. Dies würde bedeuten, dass nur eine Adresse in hexadezimaler Form ausgegeben würde.

Wollen Sie einen Manipulator verwenden, der am Anfang der Zeile die Zeilennummer mit ausgibt, so können Sie folgendermaßen schreiben:

```
// stream3.cpp
#include <iostream>
using namespace std;

ostream& countl( ostream& os ) {
   static int line = 1;
   os << line << ": ";
   line++;
   return os;
}

int main( int argc, char **argv ) {
   cout << countl << "Ein Beispiel" << endl;
   cout << countl << "Die nächste Zeile " << endl;
   cout << countl << "Und noch eine Zeile" << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
1: Ein Beispiel
2: Die nächste Zeile
3: Und noch eine Zeile
```

Manipulatoren mit Parametern

In der Header-Datei <*iomanip*> sind Standardmanipulatoren mit Parametern deklariert. Zwar sind auch gleichwertige Methoden vorhanden (siehe Tabelle 7.21), die die folgenden Arbeiten der Manipulatoren mit Parametern erledigen, doch meiner Meinung nach sind Manipulatoren noch immer die komfortablere Alternative.

Manipulator	Beschreibung
`setiosflags( ios_base::fmtflags mask );`	Setzt die in `mask` angegebenen Flags. Mehrere Flags können mit dem bitweisen ODER verknüpft werden.
`resetiosflags( ios_base::fmtflags mask );`	Löscht alle in `mask` angegebenen Flags.
`setbase ( int base );`	Setzt eine Zahlenbasis. Mögliche erlaubte Werte sind 8, 10 oder 16.
`setfill ( int c );`	Setzt das Füllzeichen auf `c` (ruft intern die Methode `fill(c)` auf).
`setprecision ( int n );`	Setzt die Genauigkeit auf `n`.
`setw ( int n );`	Setzt die Feldbreite auf `n`. Ruft dabei intern die Methode `width(n)` auf.

Tabelle 7.21 Manipulatoren mit Parameter in <iomanip>

Hierzu ein Beispiel, das diese Manipulatoren mit Parametern in der Praxis demonstriert:

```
// stream4.cpp
#include <iostream>
#include <iomanip>
using namespace std;

int main( int argc, char **argv ) {
   int ival = 100;
   float fval = 100.111;

   // ... mit Basis und Grossbuchstaben ausgeben
   cout << setiosflags(
              ios_base::showbase|ios_base::uppercase );
   cout << "Hexadezimal : " << hex << ival << endl;
```

```
   cout << "Octal       : " << oct << ival << endl;
   cout << resetiosflags(
             ios_base::showbase|ios_base::uppercase );

   cout << setbase( 8 );
   cout << "setbase( 8 ):  " << ival << endl;
   cout << setbase( 16 );
   cout << "setbase(16 ):  " << ival << endl;
   cout << setprecision( 2 )
        << setiosflags (ios_base::fixed);
   cout << "setprecision( 2 ) : " << fval << endl;
   cout << setprecision( 1 );
   cout << "setprecision( 1 ) : " << fval << endl;

   cout << setw(12);
   cout << ival <<  endl;

   cout << setfill('-') << setw(12);
   cout << ival <<  endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Hexadezimal : 0X64
Octal       : 0144
setbase( 8 ):  144
setbase(16 ):  64
setprecision( 2 ) : 100.11

setprecision( 1 ) : 100.1
          64
----------64
```

Eigene Manipulatoren mit Parametern

Eigene Manipulatoren ohne Argumente lassen sich recht einfach realisieren. Wollen Sie hingegen eigene Manipulatoren mit Parametern erstellen, ist schon etwas mehr Aufwand nötig. Hierbei genügt es nicht mehr, nur der aufzurufenden Funktion den Operator << oder >> zu übergeben. In diesem Fall sind drei Schritte erforderlich. Sie benötigen eine Hilfsfunktion, den eigentlichen Manipulator, und Sie müssen den Operator << und/oder den Operator >> zusätzlich überladen.

Hierzu wollen wir einen Manipulator `myendl(n)` erstellen, mit dem Sie die Anzahl der Zeilenvorschübe angeben können, die ausgeführt werden sollen. Zunächst benötigen Sie dazu eine Hilfsklasse, die recht universell ist und auch für

andere Beispiele verwendet werden kann. Diese Hilfsklasse stellt uns den Ausgabe-Stream-Operator zur Verfügung:

```
class ostreamHelp {
private:
   int i_;
   ostream& (*f_)(ostream&, int);
public:
   ostreamHelp(ostream& (*f)(ostream&, int), int i):
      f_(f), i_(i) {}
   friend ostream& operator<<( ostream& os, ostreamHelp m) {
      return m.f_(os, m.i_);
   }
};
```

Die Eigenschaft `f_` erhält hier einen Zeiger auf die Ausgabefunktion, die die Anzahl der Zeilenvorschübe in den Stream einfügt. Hier die Ausgabefunktion:

```
ostream& myendlHelp( ostream& os, int n) {
   while (os && n--) {
      os << '\n';
   }
   return os;
}
```

Jetzt fehlt nur noch der eigentliche Manipulator. Dieser wird über den Ausgabe-Stream-Operator von `ostreamHelp` eingefügt, der die Arbeit zwischen dem Stream und der Ausgabefunktion `myendlHelp` vornimmt:

```
ostreamHelp myendl( int n ) {
    return ostreamHelp( &myendlHelp, n );
}
```

Jetzt können Sie den Manipulator ausführen und `n` Zeilenvorschübe ausgeben. Hierzu ein komplettes Listing, das diesen Manipulator mit Parametern demonstriert:

```
// stream5.cpp
#include <iostream>
#include <iomanip>
using namespace std;

class ostreamHelp {
private:
   int i_;
   ostream& (*f_)(ostream&, int);
public:
```

```
   ostreamHelp(ostream& (*f)(ostream&, int), int i):
      f_(f), i_(i) {}
   friend ostream& operator<<( ostream& os, ostreamHelp m) {
      return m.f_(os, m.i_);
   }
};

ostream& myendlHelp( ostream& os, int n) {
   while (os && n--) {
      os << '\n';
   }
   return os;
}

ostreamHelp myendl( int n) {
    return ostreamHelp( &myendlHelp, n);
}

int main( int argc, char **argv ) {
   cout << "Eine Zeile" << endl;
   cout << "Noch eine Zeile ... " << myendl(3);
   cout << "... drei Zeilen später ... " << myendl(2);
   cout << "... zwei Zeilen später ... " << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Eine Zeile
Noch eine Zeile ...


... drei Zeilen später ...

... zwei Zeilen später ...
```

Hierzu nochmals die einzelnen Schritte, die zusammenfassen, was passiert, wenn Sie den Manipulator `myendl(n)` aufrufen:

- Es wird zunächst die globale Funktion `myendl()` mit dem Wert `n` aufgerufen. Diese Funktion legt ein Objekt an, das einen Zeiger auf die eigentliche Manipulatorfunktion und den Wert des Arguments speichert.
- Dieses Objekt wird von der Funktion `myendl()` zurückgegeben und dem Operator `<<` übergeben. Der Operator `<<` wurde ja so überladen, dass die entsprechende Manipulatorfunktion mit dem gespeicherten Argument (hier die Funktion `myendlHelp()`) aufgerufen wird.

Wenn Sie jetzt einen eigenen Manipulator mit Argumenten erstellen wollen, müssen Sie eigentlich immer nur die Arbeit der Funktion `myendlHelp()`und die Bezeichner ändern. Mit dieser Vorlage können Sie jetzt viele weitere ähnliche Manipulatoren erstellen.

Ausgabe in Felder

Die Ausgabe des Operators `<<` kann auch über Elementfunktionen mit einer bestimmten Größe erfolgen. Mit der Methode `width()` setzen Sie beispielsweise die Feldbreite der folgenden Ausgabe, und mit `fill()` können Sie angeben, wie der Rest des Feldes aufgefüllt werden soll.

Methode	Beschreibung
`int width() const;`	Liefert den aktuellen Wert der Feldbreite zurück (Standard: `0`).
`int width ( int wide );`	Setzt die Feldbreite auf den Wert `wide`. Dieser Wert gilt allerdings nur für die nächste Ausgabe bzw. Eingabe.
`int fill() const;`	Gibt das aktuelle Zeichen aus, das als Füllzeichen verwendet wird (Standard: Leerzeichen).
`int fill ( int ch );`	Verwendet das übergebene Zeichen `ch` als Füllzeichen.

Tabelle 7.22 Methoden von »ios« für die Feldbreite und das Füllzeichen

Natürlich gibt es auch hier Einstellungen für die Ausrichtung der Ausgabe. Dazu können Sie entweder die einzelnen Bits im Bit-Feld `ios::adjustfield` setzen bzw. löschen, oder aber Sie verwenden auch hierfür wieder Manipulatoren, die diese Flags setzen. Die einzelnen Flags im Bit-Feld `ios::adjustfield` finden Sie in Tabelle 7.23 und die entsprechenden Manipulatoren in Tabelle 7.24.

Flag	Beschreibung
`ios::left`	linksbündige Ausrichtung
`ios::right`	rechtsbündige Ausrichtung (Standard)
`ios::internal`	Vorzeichen linksbündig und Wert rechtsbündig im Feld

Tabelle 7.23 Flags im Bit-Feld »ios::adjustfield«

Manipulator	Beschreibung
`left`	Linksbündige Ausrichtung; setzt das Flag `ios::left`.
`right`	Rechtsbündige Ausrichtung (Standard); setzt das Flag `ios::right`.
`internal`	Vorzeichen linksbündig und Wert rechtsbündig im Feld; setzt das Flag `ios::internal`.

Tabelle 7.24 Manipulatoren für das Bit-Feld »ios::adjustfield«

Hierzu ein Beispiel, das die Ausgabe bzw. Eingabe von Feldern in der Praxis demonstrieren soll:

```
// stream6.cpp
#include <iostream>
using namespace std;

int main( int argc, char **argv ) {
   char cstr[8];
   cout << "Aktuelle Feldbreite  : "
        << cout.width() << endl;
   cout << "Aktuelles Füllzeichen: " << cout.fill() << endl;

   cout.width( 5 );
   cout.fill( '$' );
   cout << 111 << endl << 111 << endl;

   cout.fill( '#' );
   cout.width( 8 );
   cout << -111 << endl;
   cout.width( 8 );
   cout << left << -111 << endl;
   cout.width( 8 );
   cout << internal << -111 << endl;

   cout << "String eingeben (max. 8 Zeichen): ";
   // Feldbreite zur Sicherheit vorgeben
   // - nicht mehr als 8 Zeichen einlesen
   cin.width( 8 );
   cin >> cstr;
   cout << "Die Eingabe war " << cstr << endl;
   return 0;
}
```

Das Programm bei der Ausgabe:

```
Aktuelle Feldbreite  : 0
Aktuelles Füllzeichen:
$$111
111
####-111
-111####
-####111
String eingeben (max. 8 Zeichen): 123456789
Die Eingabe war 1234567
```

Unformatierte Ausgabe

In der Klasse `ostream` stehen Ihnen auch `public`-Methoden zur Verfügung, die einen Text unformatiert ausgeben bzw. einlesen (siehe Tabelle 7.25). Bei einer unformatierten Operation werden Zeichen ohne Konvertierung als `char`-Wert ausgegeben bzw. gespeichert. Dies bedeutet natürlich auch, dass gesetzte Format-Flags keinen Effekt mehr haben. Außerdem erfolgt die Aus- bzw. Eingabe nicht mehr über die überladenen Operatoren `<<` bzw. `>>`.

Methode	Beschreibung
`ostream& put( char ch );`	Gibt unformatiert das Zeichen `ch` aus.
`ostream& write( const char* s, int n );`	Gibt unformatiert die ersten `n` Zeichen des `char`-Arrays `s` aus.
`ostream& flush();`	Leert den Ausgabepuffer.

Tabelle 7.25 Methoden zur unformatierten Ausgabe

Die Anwendung dieser Methoden zur unformatierten Ausgabe erfolgt stets ohne den überladenen `<<`-Operator. Hierzu ein Beispiel:

```
// stream7.cpp
#include <iostream>
using namespace std;

int main( int argc, char **argv ) {
   char cstr[] = "Ein C-String";
   for( int i=0; cstr[i]!='\0'; i++ ) {
      cout.put(cstr[i]);
   }
   cout << endl;
   cout.write( cstr, 5 );
   cout << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Ein C-String
Ein C
```

Die Klasse »istream«

`istream` ist für das Einlesen von Daten zuständig und besitzt (erbt) die komplette Funktionalität der Klasse `ios` (wie auch schon `ostream`). Die Erweiterungen von

istream entsprechen denen von ostream. Es ändert sich nur der Datenfluss, und zusätzlich sind noch weitere Methoden definiert.

Wie auch der Operator << bei ostream arbeitet der überladene Operator >> mit den Formatierungs-Flags der Klasse ios. Auch hier bestimmen die Flags, wie die eingelesene Zeichenfolge konvertiert werden soll. So können Sie zum Beispiel eine Ziffernfolge dezimal, hexadezimal oder oktal abspeichern.

Eine besondere Fähigkeit von istream ist, dass der Operator >> für die formatierte Ausgabe der elementaren Datentypen, C-Strings, Manipulatoren und einige Zeiger überladen ist (siehe Tabelle 7.26). Der Stream cin ist zum Beispiel ein Objekt der Klasse istream. Beispielsweise wird mit der Anweisung

```
int val;
cin >> val ;
```

die Methode operator>>(int) von istream aufgerufen. Mit

```
char ch
cin >> ch;
```

würde die Methode operator>>(char) aufgerufen.

ostream& operator>>(TYP)	Beschreibung
char&, unsigned char&, signed char&	Ein Zeichen wird eingelesen.
short&, unsigned short&, int&, unsigned int&, long&, unsigned long&	Eine Ganzzahl wird eingelesen. Wird eine führende 0 angegeben oder ein führendes 0x (bzw. 0X), wird die Zahl als oktaler bzw. hexadezimaler Wert eingelesen, ansonsten dezimal.
float&, double&, long double&	Eine Gleitpunktzahl wird eingelesen. Hierbei wird sowohl die wissenschaftliche Notation als auch die Festpunktdarstellung akzeptiert.
char*, signed char*, unsigned char*	Liest eine Zeichenkette in ein char-Array ein und hängt das Stringende-Zeichen hinten an. Beim Auftreten eines Whitespace-Zeichens (Leerzeichen, Tabulator, Newline) wird das Einlesen beendet.
void&	Liest eine Adresse ein und speichert dies im Zeiger ab.
streambuf*	Kopiert die Zeichen aus dem Puffer des Streams in den übergebenen Puffer. Die Übertragung der Zeichen wird beendet, wenn EOF oder ein Fehler aufgetreten ist.
ios& (*pf)(ios&), istream& (*pf) (istream&)	Ruft den durch pf adressierten Manipulator von ios bzw. istream auf.

Tabelle 7.26 Auch der >>-Operator ist überladen für die formatierte Eingabe.

Manipulatoren

In der Klasse `istream` ist nur ein Manipulator mit `ws` definiert. Mit diesem ist es möglich, Zwischenraumzeichen wie Leerzeichen, Tabulator oder Newline aus dem Eingabepuffer zu entfernen.

Manipulator	Beschreibung
`ws`	Entfernt die Zwischenraumzeichen (Leerzeichen, Tabulator, Newline) aus dem Eingabepuffer bis zum ersten Zeichen, das keins ist.

Tabelle 7.27 Manipulator für die Klasse »istream«

Hierzu ein Beispiel, das zeigt, wie der Manipulator `ws` in der Praxis verwendet wird:

```
// stream8.cpp
#include <iostream>
using namespace std;

int main( int argc, char **argv ) {
   char name[20];
   cout << "Der Nachname bitte: ";
   cin >> ws;
   cin.getline( name, 20 );
   cout << "Der Nachname lautete " << name << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Der Nachname bitte: ↵      Wolf
Der Nachname lautete Wolf
```

Unformatierte Eingabe

Auch für die Eingabe liegt eine Reihe von `public`-Methoden vor, mit denen man unformatiert einzelne Zeichen und Zeichenfolgen von der Standardeingabe lesen kann (siehe Tabelle 7.28). Hierbei wird, wie schon bei der Ausgabe, das Zeichen ohne Umwandlung als `char`-Wert gespeichert. Die Format-Flags haben auch hier keinen Effekt mehr, und der überladene Operator `>>` wird nicht mehr benötigt.

Methode	Beschreibung
`int get();`	Liest unformatiert ein Zeichen ein und gibt es zurück.
`istream& get( char& ch );`	Liest unformatiert ein Zeichen ein und speichert es in `ch`.

Tabelle 7.28 Methoden von »istream« zur formatierten Eingabe

Methode	Beschreibung
`istream& get( char* s, int len, char delim= '\n' );`	Liest `len-1` Zeichen unformatiert ein und speichert die Zeichen im Array `s`. Es wird allerdings höchstens bis zum nächsten `delim` gelesen. Das Stringende-Zeichen wird hinten angefügt.
`istream& getline( char* s, int len, char delim= '\n' );`	Liest unformatiert alle Zeichen bis zum nächsten `delim`-Zeichen (oder maximal `len-1` Zeichen) ein und speichert diese Zeichen im Array `s`. Das `delim`-Zeichen wird nicht im Array gespeichert. Am Ende wird noch das Stringende-Zeichen hinzugefügt.
`istream& read( char* s, int n );`	Liest unformatiert maximal `n` Zeichen (oder bis höchstens `EOF`) ein und speichert diese im Array `s`.
`int readsome( char* s, int n );`	Liest unformatiert `n` Zeichen in das Array `s` ein. Hierbei werden allerdings nur so viele Zeichen eingelesen, wie im Puffer stehen. Zurückgegeben wird die Anzahl der gelesenen Zeichen.
`istream& ignore( int n=1, char delim = EOF );`	Überliest max `n` Zeichen aus dem Eingabepuffer bzw. höchstens bis zum Zeichen `delim`. Das `delim`-Zeichen wird hierbei aus dem Puffer entfernt.
`istream& putback( char ch );`	Schiebt das Zeichen `ch` wieder in den Puffer zurück, so dass dieses beim nächsten Einlesen wieder zur Verfügung steht.
`istream& unget();`	Schiebt das zuletzt eingelesene Zeichen wieder in den Puffer zurück.
`int gcount();`	Gibt die Anzahl der Zeichen zurück, die zuletzt unformatiert eingelesen wurden.

Tabelle 7.28 Methoden von »istream« zur formatierten Eingabe (Forts.)

Die unformatierte Eingabe wird bevorzugt verwendet, wenn es um die Ein-/Ausgabe von Binär-Dateien geht, weil Sie hiermit auf jedes einzelne Byte zugreifen können.

Wenn kein Zeichen eingelesen werden konnte oder das Dateiende erreicht wurde, ist es – wie immer – unerlässlich, die Status-Flags von `ios` zu überprüfen. Wie Sie das machen, erfahren Sie ein paar Seiten weiter. Bei den Methoden, die einen Integer zurückgeben, können Sie den Rückgabewert auf `EOF` überprüfen. Trat `EOF` auf, ist entweder ein Fehler aufgetreten, oder das Dateiende wurde erreicht.

Hierzu ein Beispiel, das u. a. Datei-Streams verwendet – siehe Abschnitt 7.2.2, »Klassen für Datei-Streams (File-Streams)« – und zeigt, wie Sie die unformatierte Eingabe einsetzen können.

```
// stream9.cpp
#include <iostream>
#include <fstream>
using namespace std;

int main( int argc, char **argv ) {
   char ch;
   char nam[20];
   if( argc != 3 ) {
      cout << "Verwendung " << argv[0]
           << "originalDatei kopieDatei" << endl;
      return 1;
   }
   ifstream in(argv[1]);
   ofstream out(argv[2]);
   // Unformatiert byteweise kopieren
   while( ch = in.get( ) ) {
      if( ch == EOF ) {
         cout << "Dateiende erreicht oder Fehler ..."
              << endl;
         break;
      }
      out.put(ch);
   }
   cout << "Name eingeben: ";
   cin.getline( nam, 20 );
   cout << "Es wurden " << cin.gcount()
        << " Zeichen eingelesen" << endl;

   cout << "Nochmals Name eingeben: ";
   cin.ignore( 3 );
   cin.getline( nam, 20 );
   cout << "Es wurden " << cin.gcount()
        << " Zeichen eingelesen" << endl;
   cout << nam << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
$ ./stream9 stream9.cpp stream9.bak
Dateiende erreicht oder Fehler ...
Name eingeben: J.Wolf
Es wurden 7 Zeichen eingelesen
Nochmals Name eingeben: J.Wolf
Es wurden 4 Zeichen eingelesen
olf
```

Ein-/Ausgabe-Stream verbinden

Sollen die Streams für die Ein-/Ausgabe-Operationen synchronisiert werden, haben Sie die Möglichkeit, einen Eingabe-Stream mit einem Ausgabe-Stream zu verbinden. Die Klasse `ios` stellt mit `tie()` eine solche Methode zur Verfügung.

```
ostream* tie() const;
ostream* tie( ostream* stream );
```

Wird diese Methode ohne Argumente aufgerufen, wird die Adresse des verbundenen Streams zurückgegeben. Gibt es keinen solchen Stream, wird `NULL` zurückgegeben. Mit einem Argument wird der Stream mit dem als Argument angegebenen Stream verbunden. Wird ein Eingabe-Stream mit einem Ausgabe-Stream verbunden, wird jeweils vor dem Einlesen der Puffer des Ausgabe-Streams entleert. Das bedeutet, dass die Eingabeaufforderung erscheint, bevor das System für das Lesen der Eingabe blockiert. Dies entspricht im Grunde einem Aufruf von `flush()`.

Per Standard ist die Standardeingabe mit der Standardausgabe wie folgt verbunden:

```
// vordefinierte Verbindung
cin.tie (&cout);
wcin.tie (&wcout);
```

Dies stellt sicher, dass immer die Aufforderung vor der Eingabe gemacht wird:

```
int val;
cout << "Einen Wert bitte: ";
cin >> val;
```

Hier wird vor der Ausgabe noch implizit die Methode `flush()` aufgerufen, bevor der Wert `val` eingelesen wird. So liefert ein Aufruf wie

```
ostream* PtrStr = cin.tie();
```

standardmäßig immer die Adresse von `cout`. Per Standard sind die Objekte `cin`, `cerr` und `clog` mit `cout` verbunden.

Hierzu noch ein weiteres Beispiel, das zeigt, das `tie()` nicht nur auf die Ein-/Ausgabe-Streams von `ostream` bzw. `istream` beschränkt ist. Hier wird nochmals `ofstream` verwendet, worauf in Abschnitt 7.2.2, »Klassen für Datei-Streams (File-Streams)«, noch näher eingegangen wird:

```
// stream10.cpp
#include <iostream>
#include <fstream>
using namespace std;
```

```
int main( int argc, char **argv ) {
   ostream *prevstr;
   ofstream filestr;
   // Datei-Stream öffnen
   filestr.open ("datei.txt");

   // Ausgabe auf cout, da Standard
   *cin.tie() << "Die kommt in cout" << endl;
   // Ausgabe mit filestr verbinden
   // Adresse von cout zum Wiederherstellen sichern
   prevstr = cin.tie (&filestr);
   // Dies wird in die Datei "datei.txt" geschrieben
   *cin.tie() << "Und dies kommt in die Datei" << endl;
   // Alten Zustand wiederherstellen
   cin.tie (prevstr);
   *cin.tie() << "Jetzt wieder cout" << endl;
   // Datei-Stream wieder schließen
   filestr.close();
   return 0;
}
```

Status und Flags von Ein-/Ausgabe-Streams

Um die Eigenschaften von Streams abzufragen, ob zum Beispiel ein Fehler aufgetreten ist, werden Flags verwendet. Im Grunde werden diese Flags in drei Bereiche aufgeteilt:

- Status-Flags
- Formatierungs-Flags
- Flags zum Öffnen einer Datei

Status-Flags

Der aktuelle Status eines Streams wird in einem Statuswort gespeichert. Folgende Status-Flags sind in `ios` mit dem Typ `iostate` definiert:

Flag	Beschreibung
`ios::goodbit`	Kein Fehler vorhanden.
`ios::eofbit`	Das Dateiende wurde erreicht.
`ios::failbit`	Bei der letzten Operation ist ein Fehler aufgetreten.
`ios::badbit`	Der Stream ist defekt.

Tabelle 7.29 Status-Flags aus der Klasse »ios«

Der Unterschied zwischen `ios::failbit` und `ios::badbit` ist auf den ersten Blick nicht ersichtlich. `ios::badbit` ist gesetzt, wenn ein nicht behebbarer Fehler aufgetreten ist – wie beim Schreiben in einer Datei. `ios::failbit` hingegen wird gesetzt, wenn bei der Ein-/Ausgabe etwas falsch gelaufen ist. Dies kann zum Beispiel der Fall sein, wenn ein Buchstabe statt einer Zahl eingegeben wurde.

Der Zugriff auf die einzelnen Flags wird mit einfachen Methoden realisiert, die in der Tabelle 7.30 aufgelistet sind.

Methode	Beschreibung
`bool good() const;`	Gibt `true` zurück, wenn kein Fehler aufgetreten ist, also keine Status-Bit gesetzt sind.
`bool eof() const;`	Gibt `true` zurück, wenn `ios::eofbit` gesetzt ist.
`bool fail() const;`	Gibt `true` zurück, wenn `ios::failbit` oder `ios::badbit` gesetzt ist.
`bool bad() const;`	Gibt `true` zurück, wenn `ios::badbit` gesetzt ist.
`operator void*() const;`	Gibt ungleich `NULL` zurück, wenn `ios::failbit` und `ios::badbit` nicht gesetzt sind.
`bool operator!() const;`	Gibt `true` zurück, wenn `ios::failbit` oder `ios::badbit` gesetzt ist.
`iostate rdstate() const;`	Liefert das Statuswort des Streams zurück.
`void clear( iostate state = goodbit );`	Verwendet das übergebene Argument als neues Statuswort. Ohne Argument wird `ios::goodbit` gesetzt.
`void setstate( iostate state );`	Setzt das Statuswort des Streams mit dem Funktionsaufruf `clear(rdstate() | state)`.

Tabelle 7.30 Methoden von »ios« zur Statusabfrage eines Streams

Hierzu wieder ein Listing, das in der Praxis zeigt, wie man den Status mit Hilfe der in Tabelle 7.30 vorgestellten Methoden abfragen bzw. verändern kann:

```
// stream11.cpp
#include <iostream>
#include <fstream>
using namespace std;

void checkStatus( ios& stream, const char* str );

int main( int argc, char **argv ) {
   int ival;
   checkStatus( cout, "cout" );
   checkStatus( cin, "cin" );
   cout << "Bitte eine Zahl eingeben: ";
   cin >> ival;
   checkStatus( cin, "cin" );
```

```
   ifstream is;
   is.open ("test.txt");
   checkStatus( is, "ifstream is" );

   ifstream is2;
   is2.open ("test.txt");

   // ... oder alternativ mit rdstate()
   if ( (is2.rdstate() & ifstream::failbit ) != 0 ) {
       cout << "ifstream is2:Fehler bei der Ein-/Ausgabe!"
            << endl;
   }
   // Statuswort zurücksetzen
   is2.clear();
   checkStatus( is2, "ifstream is2" );

   // Statuswort von cout verändern ...
   cout.setstate( ios::badbit );
   checkStatus( cout, "cout" );

   // So gehts auch dank "bool operator!() const;"
   cout << "Bitte eine Zahl eingeben: ";
   if( !(cin >> ival) ) {
      checkStatus( cin, "cin");
   }
   return 0;
}

void checkStatus( ios& stream, const char* str="" ) {
   if( stream.good() == true ) {
      cerr << str << ":Status-Flags in Ordnung" << endl;
   }
   else if( stream.bad() == true ) {
      cerr << str << ":Fataler Fehler aufgetreten" << endl;
   }
   else if( stream.fail() == true ) {
      cerr << str << ":Fehler bei der Ein-/Ausgabe!"
           << endl;
   }
   else if( stream.eof() == true ) {
      cerr << str << ":Dateiende erreicht" << endl;
   }
   // Alles wieder zurücksetzen
   stream.clear();
}
```

Das Programm bei der Ausführung:

```
cout:Status-Flags in Ordnung
cin:Status-Flags in Ordnung
Bitte eine Zahl eingeben: 4
cin:Status-Flags in Ordnung
ifstream is:Fehler bei der Ein-/Ausgabe!
ifstream is2:Fehler bei der Ein-/Ausgabe!
ifstream is2:Status-Flags in Ordnung
cout:Fataler Fehler aufgetreten
Bitte eine Zahl eingeben: X
cin:Fehler bei der Ein-/Ausgabe!
```

Format-Flags

Auf die Formatierung der Ein- bzw. Ausgabe mit den Operatoren << bzw. >> wurde bereits indirekt eingegangen. Allerdings wurde hierbei nicht direkt mit den Format-Flags gearbeitet. Die Format-Flags sind ebenfalls in ganzzahligen Variablen gespeichert, die in `ios` mit `fmtflags` definiert sind (was gewöhnlich ein `long`-Wert ist). Jedes dieser Format-Flags wird zu einem Bit-Feld zusammengefasst. Folgende drei Bit-Felder sind in `ios` definiert:

Bit-Feld	Beschreibung
`ios::basefield`	das Zahlensystem für ganzzahlige Werte
`ios::floatfield`	Darstellung von Gleitkommazahlen
`ios::adjustfield`	Ausrichtung im Ausgabefeld

Tabelle 7.31 Die einzelnen Bit-Felder in »ios«

Jetzt folgen jeweils die einzelnen Flags, die in diesen Bit-Feldern definiert sind (siehe Tabellen 7.32 bis 7.35).

Flag (ios::basefield)	Beschreibung
`ios::oct`	Ein- und Ausgabe erfolgen oktal.
`ios::dec`	Ein- und Ausgabe erfolgen dezimal (Standard).
`ios::hex`	Ein- und Ausgabe erfolgen hexadezimal.

Tabelle 7.32 Flags im Bit-Feld »ios::basefield«

Flag (ios::floatfield)	Beschreibung
`ios::fixed`	Gleitpunktzahlen werden als Festpunktzahl dargestellt.
`ios::scientific`	Gleitpunktzahlen werden als exponentielle Schreibweise dargestellt.

Tabelle 7.33 Flags im Bit-Feld »ios::floatfield«

Flag (ios::floatfield)	Beschreibung
`ios::showpoint`	Bei der Ausgabe von Gleitpunktzahlen immer den Dezimalpunkt mit ausgeben.

Tabelle 7.33 Flags im Bit-Feld »ios::floatfield« (Forts.)

Flag (ios::adjustfield)	Beschreibung
`ios::left`	Ausrichtung erfolgt linksbündig im Feld.
`ios::right`	Ausrichtung erfolgt rechtsbündig im Feld (Standard).
`ios::internal`	Vorzeichen linksbündig, Wert rechtsbündig

Tabelle 7.34 Flags im Bit-Feld »ios::adjustfield«

Flag (ios::adjustfield)	Beschreibung
`ios::showbase`	Basis bei der Ausgabe von Ganzzahlen angeben; bei oktalen Zahlen mit führender `0` und bei hexadezimalen Werten `0x` bzw. `0X`.
`ios::showpos`	Positive Werte werden mit dem +-Vorzeichen ausgegeben.
`ios::uppercase`	Bei der Ausgabe von Ganzzahlen werden Großbuchstaben verwendet. Beispielsweise wird bei einem hexadezimalen Wert `0X` verwendet oder bei exponentieller Darstellung `E`.

Tabelle 7.35 Flags für die Ausgabe von Zahlen

Auf viele dieser Flags haben Sie bereits bei den Manipulatoren zugegriffen – was erheblich komfortabler ist als die hier gleich vorgestellten Methoden. Zunächst ein Überblick über die Elementfunktionen für den Zugriff auf diese Format-Flags:

Methode	Beschreibung
`fmtflags flags();`	Gibt alle Format-Flags zurück. Wird gewöhnlich zum Sichern der Format-Flags verwendet.
`fmtflags flags(fmtflags flags);`	Setzt eine komplette Flag-Variable. Wird gewöhnlich verwendet, um die gesicherte Flag-Variable wiederherzustellen.
`fmtflags setf(fmtflags flags);`	Setzt einzelne Flags in der Format-Variablen.

`fmtflags setf(`
`  fmtflags flags,`
`  fmtflags bitfield );`
	Löscht alle Flags im entsprechenden `bitfield` und setzt anschließend einzelne Flags.

`fmtflags unsetf(fmtflags flags);`	Löscht einzelne Flags in der Format-Variablen.

Tabelle 7.36 Methoden für den Zugriff auf Format-Flags

Der Einsatz dieser Methoden ist relativ einfach. Mit der Methode `flags()` werden die Flags gewöhnlich eingelesen und gesichert und anschließend auch wie-

derhergestellt. Mit `setf()` werden die einzelnen Flags gesetzt und mit `unsetf()` wieder gelöscht. Hier ein einfaches Beispiel für den Einsatz dieser Methoden:

```
// stream12.cpp
#include <iostream>
#include <fstream>
using namespace std;

int main( int argc, char **argv ) {
   int ival = 100;
   float fval = 11.11;
   // Flags sichern
   ios::fmtflags backupFlags = cout.flags();

   // Im Bitfeld ios::basefield alle Flags löschen -
   // und das Flag hex setzen ...
   cout.setf( ios::hex, ios::basefield );
   // Im Bitfeld ios::floatfield alle Flags löschen -
   // und das Flag scientific setzen
   cout.setf( ios::scientific, ios::floatfield );
   // Basis anzeigen
   cout.setf( ios::showbase );

   cout << "ival : " << ival << endl;
   cout << "fval : " << fval << endl;

   // scientific löschen
   cout.unsetf ( ios::scientific );
   cout << "ival : " << ival << endl;
   cout << "fval : " << fval << endl;

   // Alles wiederherstellen
   cout.flags( backupFlags );
   cout << "ival : " << ival << endl;
   cout << "fval : " << fval << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
ival : 0x64
fval : 1.111000e+001
ival : 0x64
fval : 11.11
ival : 100
fval : 11.11
```

7.2.2 Klassen für Datei-Streams (File-Streams)

Die File-Streams `ifstream`, `ofstream` und `fstream` basieren auf den Grundlagen der Streams `ios`, `istream` und `ostream` (siehe Abbildung 7.2). Alle diese Streams werden zur Dateiverarbeitung verwendet und sind in der Header-Datei <*fstream*> deklariert.

`ifstream` ist eine `public`-Vererbung der Klasse `istream` und wird zum Lesen von Dateien eingesetzt. `ofstream` wiederum ist eine `public`-Vererbung der Klasse `ostream` und wird zum Schreiben von Dateien verwendet. `fstream` ist eine Vererbung der Klasse `iostream` und ermöglicht sowohl das Schreiben als auch das Lesen von Dateien.

Da all diese Klassen entsprechende Basisklassen haben, die hier bereits beschrieben wurden, können auch alle Lese- und Schreiboperationen dieser Basisklassen verwendet werden.

fstream

Die Klasse `fstream` entsteht durch eine `public`-Vererbung der Klasse `iostream`. Somit stehen alle formatierten Ein-/Ausgabefunktionalitäten auch in `fstream` zur Verfügung. `fstream` wird gewöhnlich verwendet, wenn in einem Programm sowohl aus einer Datei gelesen als auch geschrieben werden soll (die Ein-/Ausgabeoperatoren << und >>).

Vor einem Zugriff auf eine Datei müssen Sie diese mit der Methode `open()` öffnen. Dies kann theoretisch auch zusammen mit dem Konstruktor der Stream-Klasse gemacht werden. Wenn Sie mit der Datei fertig sind, müssen Sie diese ordnungsgemäß mit `close()` schließen, damit das Objekt wieder zerstört wird. Hierzu ein Überblick über die vorhandenen Methoden der `fstream`-Klasse:

Methode	Beschreibung
`fstream();`	Legt einen File-Stream und einen Dateipuffer `filebuf` an, ohne eine Datei zu öffnen.

`explicit fstream(`
`  const char* s,`
`  openmode m=ios::in|ios::out);`
	Konstruktor; öffnet die Datei `s` im angegebenen Modus (`openmode m`). Standardmäßig ist der Modus zum Lesen und Schreiben geöffnet.

`void open(`
`  const char* s,`
`  openmode m=ios::in|ios::out);`
	Öffnet die angegebene Datei `s` im Modus `m`. Fehlt der Modus, wird die Datei zum Lesen und Schreiben geöffnet.

`bool is_open();`	Gibt `true` zurück, wenn eine Datei bereits geöffnet wurde.

Tabelle 7.37 »public«-Methoden der Klasse »fstream«

Methode	Beschreibung
`filebuf rdbuf();`	Gibt die Adresse des Stream-Puffers zurück.
`void close();`	Schließt die akuell geöffnete Datei.

Tabelle 7.37 »public«-Methoden der Klasse »fstream« (Forts.)

Dem Konstruktor und der Methode `open()` kann man als Argument auch den Öffnungsmodus übergeben. In der Klasse `ios` sind dabei verschiedene Modi vom Typ `ios::openmode` definiert, die in der folgenden Tabelle (7.38) aufgelistet werden.

Flag	Beschreibung
`ios::in`	Datei wird zum Lesen geöffnet.
`ios::out`	Datei wird zum Schreiben geöffnet.
`ios::app`	Vor jeder Schreiboperation wird der Zeiger auf das Dateiende gesetzt.
`ios::trunc`	Beim Öffnen einer Datei die Länge auf 0 setzen. Achtung, löscht den Inhalt der Datei!
`ios::ate`	Datei wird geöffnet und der Zeiger auf das Dateiende gesetzt.
`ios::binary`	Lese- und Schreiboperationen im Binärmodus ausführen

Tabelle 7.38 Öffnungsmodus der Klasse »ios«

Mehrere Flags können, sofern dies sinnvoll erscheint, mit dem Bit-Operator `|` verknüpft werden. Hierzu ein einfaches Beispiel, das diese Methoden der Klasse `fstream` in der Praxis zeigen soll:

```
// fstream1.cpp
#include <iostream>
#include <fstream>
using namespace std;

int main( int argc, char **argv ) {
   fstream f1, f3;
   // Datei zum Lesen und Schreiben öffnen
   fstream f2("test.txt");
   // Datei zum Lesen und Schreiben öffnen
   f1.open("test2.txt");
   // Datei nur zum Lesen öffnen
   f3.open("test3.txt", ios::in);
   // ios::failbit überprüfen
   if( ! (f1) ) {
      cerr << "Konnte Datei test2.txt nicht öffnen!"
```

```
            << endl;
      //return 1;
   }
   // ... oder so überprüfen ...
   if( f2.fail() == true ) {
      cerr << "Konnte Datei test.txt nicht öffnen!" << endl;
      //return 1;
   }
   // ... oder auch so ...
   if( f2.is_open() != true ) {
      cerr << "Konnte Datei test3.txt nicht öffnen!"
           << endl;

      //return 1;
   }
   f1 << "Dies kommt in die Datei test2.txt" << endl;
   f2 << "Dies kommt in die Datei test.txt" << endl;
   // Achtung - Datei wurde nur zum Lesen geöffnet!
   f3 << "Dies kommt nicht in die Datei test3.txt" << endl;

   if( f3.fail() == true ) {
      cerr << "Fehler beim Schreiben in test3.txt" << endl;
      f3.clear();
      //return 1;
   }
   // File-Streams wieder schließen
   f1.close();
   f2.close();
   f3.close();
   return 0;
}
```

Wenn alles gutgegangen ist und die Dateien *test.txt*, *test2.txt* und *test3.txt* existieren, sollte in den Dateien *test.txt* und *test2.txt* eine Zeile zu finden sein. In *test3.txt* hingegen konnte nicht geschrieben werden, da diese Datei ja im Lesemodus (`ios:in`) geöffnet wurde.

[»] Hinweis

Natürlich setzt ein schreibender Zugriff auch voraus, dass Sie entsprechende Zugriffsrechte auf diese Datei haben.

Hinweis zum Dateinamen [«]

Der Pfadname ist ein C-String. Achten Sie bitte darauf, dass Sie unter MS-Systemen als Pfadtrenner zweimal einen Backslash verwenden, weil dieser sonst in einem String als Sonderzeichen interpretiert würde. MS Windows akzeptiert aber mittlerweile auch – wie bei Linux/UNIX üblich – einen Schrägstrich als Pfadtrenner.

»ifstream« und »ofstream«

Benötigen Sie hingegen nur einen File-Stream zum Lesen von Dateien, können Sie auch nur die Klasse `ifstream` bzw. zum Schreiben von Dateien nur die Klasse `ofstream` verwenden. `ifstream` ist eine `public`-Vererbung der Klasse `istream`, und `ofstream` wiederum ist eine `public`-Vererbung der Klasse `ostream`.

Beide Klassen besitzen dieselben `public`-Methoden wie die Klasse `fstream` (siehe Tabelle 7.39), nur dass die Konstruktoren entsprechend anders lauten und der Standardmodus entsprechend angegeben ist; hierzu finden Sie die Konstruktoren und die öffentliche Methode `open()` der Klasse `ofstream` in der Tabelle 7.39. Die anderen öffentlichen Methoden können Sie der Tabelle 7.37 zu `fstream` entnehmen.

Methode	Beschreibung
`ofstream();`	Legt einen File-Stream zum Schreiben und einen Dateipuffer `filebuf` an, ohne eine Datei zu öffnen.

`explicit ofstream(`
`  const char* s,`
`  openmode m = ios::out |`
`  ios::trunc);`
	Konstruktor; öffnet die Datei `s` im angegebenen Modus (`openmode m`). Standardmäßig ist der Modus zum Schreiben geöffnet, und die Datei wird auf die Länge `0` gekürzt.

`void open(`
`  const char* s,`
`  openmode m = ios::out |`
`  ios::trunc);`
	Öffnet die angegebene Datei `s` im Modus `m`. Fehlt der Modus, wird die Datei zum Schreiben geöffnet und die Datei auf die Länge `0` gekürzt.

Tabelle 7.39 Konstruktoren und Methoden von »ofstream«

Ebenso sehen die Gegenstücke zu den `public`-Methoden der Klasse `ofstream` aus (siehe Tabelle 7.40). Alle in der folgenden Tabelle nicht erwähnten Methoden zu `ifstream` entnehmen Sie bitte der Tabelle 7.37 zu `fstream`.

Methode	Beschreibung
`ifstream();`	Legt einen File-Stream zum Lesen und einen Dateipuffer `filebuf` an, ohne eine Datei zu öffnen.

Tabelle 7.40 Konstruktoren und Methoden von »ifstream«

Methode	Beschreibung
`explicit ifstream( const char* s, openmode m = ios::in );`	Konstruktor; öffnet die Datei `s` im angegebenen Modus (`openmode m`). Standardmäßig ist der Modus zum Lesen geöffnet.
`void open( const char* s, openmode m = ios::in );`	Öffnet die angegebene Datei `s` im Modus `m`. Fehlt der Modus, wird die Datei zum Lesen geöffnet.

Tabelle 7.40 Konstruktoren und Methoden von »ifstream« (Forts.)

Die Anwendung ist ähnlich derjenigen, die im Beispiel zu `fstream` demonstriert wurde. Trotzdem ein kleines Listing zu `ofstream` und `ifstream`:

```
// fstream2.cpp
#include <iostream>
#include <fstream>
using namespace std;

int main( int argc, char **argv ) {
   int value = 100;
   int val2;
   ifstream f1;
   // Datei zum Schreiben öffnen
   ofstream f2("test.txt");
   if( f2.is_open() != true ) {
      cerr << "Konnte Datei test.txt nicht öffnen!" << endl;
      //return 1;
   }
   // Als Hexzahl schreiben
   f2 << hex <<  value;
   f2.close();
   // Datei zum Lesen öffnen
   f1.open("test.txt");
   if( f1.is_open() != true ) {
      cerr << "Konnte Datei test.txt nicht öffnen!" << endl;
      //return 1;
   }
   f1 >> val2;
   cout << "Wert in test.txt: " << val2 << endl;
   f1.close();
   return 0;
}
```

Wenn Sie das Programm ausführen, wird zunächst der Wert von `value` in die Datei *test.txt* als hexadezimale Ganzzahl geschrieben. Anschließend wird dieser Wert wieder ausgelesen und auf dem Bildschirm ausgegeben.

Datenstrom

Im Beispiel *fstream2.cpp* konnten Sie sehen, dass sich die Stream-Objekte für Dateien genauso verwenden lassen, wie Sie dies von `cin` bzw. `cout` kennen. Natürlich können Sie auch alle bekannten Manipulatoren und Formatierungen verwenden, die Sie bisher kennengelernt haben:

```
// fstream3.cpp
#include <iostream>
#include <fstream>
using namespace std;

int main( int argc, char **argv ) {
   int value = 100;
   // Datei zum Schreiben öffnen
   ofstream f1("test.txt");
   if( f1.is_open() != true ) {
      cerr << "Konnte Datei test.txt nicht öffnen!" << endl;
      //return 1;
   }
   // Füllzeichen '*' verwenden
   f1.fill('*');
   // Feldbreite 8 Zeichen
   f1.width( 8 );
   // Als Hexzahl schreiben
   f1 << hex <<  value;
   f1.close();
   return 0;
}
```

In der Textdatei *test.txt* würde jetzt Folgendes stehen:

```
******64
```

Entsprechend verläuft dies mit dem Eingabeoperator `>>` in gleicher Weise wie mit `cin`. Dabei wird die Datei gelesen, als käme die Eingabe von der Tastatur. Allerdings haben Sie auch in diesem Fall mit dem `cin`-typischen Whitespace-Zeichen zu tun. Auch hierbei werden Leerzeichen, Tabulatoren und Newline-Zeichen als Trenner der Eingabe verwendet. Somit würde

```
char buffer[100];
ifstream f1("test.txt");
...
```

```
f1.width(100);
f1 >> buffer;
```

nur Zeichen bis zum ersten Auftreten eines Whitespace-Zeichens einlesen.

Zeichenweise Ein-/Ausgabe

Für das zeichenweise Lesen und Schreiben von Dateien können Sie die beiden Methoden `put()` und `get()` verwenden. Dabei bezieht sich jede Ein-/Ausgabeoperation auf die aktuelle Position des Schreib-/Lesezeigers. Mit jedem gelesenen Zeichen wird der `get`-Zeiger um 1 und mit jedem geschriebenen Zeichen wird der `put`-Zeiger um 1 inkrementiert. Ist das Dateiende erreicht, wird das `EOF`-Bit gesetzt.

Das folgende Listing enthält zwei Funktionen. Mit der ersten Funktion können Sie zeichenweise aus einer Datei lesen und diese zeichenweise auf die Standardausgabe ausgeben. Mit der zweiten Funktion können Sie eine Datei zeichenweise kopieren:

```
// fstream4.cpp
#include <iostream>
#include <fstream>
using namespace std;

void charOutput( ifstream& stream );
void copyChar4Char( ifstream& istream, ofstream& ostream );

int main( int argc, char **argv ) {
   // Datei zum Lesen öffnen
   ifstream f1("schiller.txt");
   ifstream f2("schiller.txt");
   ofstream f3("backSchiller.txt");
   if( f1.is_open() != true ) {
      cerr << "Konnte Datei schiller.txt nicht öffnen!"
           << endl;
      //return 1;
   }
   if( f2.is_open() != true ) {
      cerr << "Konnte Datei schiller.txt nicht öffnen!"
           << endl;
      //return 1;
   }
   if( f3.is_open() != true ) {
      cerr << "Konnte Datei backSchiller.txt nicht öffnen!"
           << endl;
```

```
      //return 1;
   }
   // Zeichenweise einlesen und ausgeben
   charOutput( f1 );
   // Zeichenweise kopieren
   copyChar4Char( f2, f3 );
   f1.close();
   f2.close();
   f3.close();
   return 0;
}

// Zeichenweise auslesen
void charOutput ( ifstream& stream ) {
   char ch;
   while( stream.get(ch) ) {
      if( stream.fail() ) {
         cout << "Unerwarteter Lesefehler!" << endl;
         break;
      }
      cout.put(ch);
   }
   cout << endl;
}

// Zeichenweise kopieren
void copyChar4Char( ifstream& istream, ofstream& ostream ) {
   char ch;
   while( istream.get(ch) ) {
      if( istream.fail() ) {
         cout << "Unerwarteter Lesefehler!" << endl;
         break;
      }
      ostream.put(ch);
      if( ostream.fail() ) {
         cout << "Unerwarteter Schreibfehler!" << endl;
         break;
      }
   }
   cout << endl;
}
```

Das Programm bei der Ausführung:

```
Rasch tritt der Tod den Menschen an,
Es ist ihm keine Frist gegeben,
Es stürzt ihn mitten in der Bahn,
Es reisst ihn fort vom vollen Leben,
Bereitet oder nicht, zu gehen,
Er muss vor seinem Richter stehen!
```

Außerdem wurde der Inhalt der Datei *schiller.txt* in die Datei *backSchiller.txt* kopiert.

Zeilenweise Ein-/Ausgabe

Zum zeilenweisen Auslesen einer Datei wird die Methode `getline()` verwendet. Dieser Methode wird als erster Parameter ein Zeiger auf ein `char`-Array übergeben. Mit dem zweiten Parameter geben Sie die maximale Anzahl von Zeichen an, die in den Puffer (erstes Argument) passen.

Hierzu dasselbe Beispiel wie schon im Listing *fstream4.cpp*, nur dass jetzt die komplette Ein-/Ausgabe zeilenweise abläuft:

```
// fstream5.cpp
#include <iostream>
#include <fstream>
using namespace std;

void lineOutput( ifstream& stream );
void copyLine4Line( ifstream& istream, ofstream& ostream );

int main( int argc, char **argv ) {
   // Datei zum Lesen öffnen
   ifstream f1("schiller.txt");
   ifstream f2("schiller.txt");
   ofstream f3("backSchiller.txt");
   if( f1.is_open() != true ) {
      cerr << "Konnte Datei schiller.txt nicht öffnen!"
           << endl;
      //return 1;
   }
   if( f2.is_open() != true ) {
      cerr << "Konnte Datei schiller.txt nicht öffnen!"
           << endl;
      //return 1;
   }
```

```
   if( f3.is_open() != true ) {
      cerr << "Konnte Datei backSchiller.txt nicht öffnen!"
           << endl;
      //return 1;
   }
   // Zeichenweise einlesen und ausgeben
   lineOutput( f1 );
   // Zeichenweise kopieren
   copyLine4Line( f2, f3 );
   f1.close();
   f2.close();
   f3.close();
   return 0;
}

// Zeilenweise auslesen
void lineOutput ( ifstream& stream ) {
   char buffer[512];
   int cnt=1;
   while( stream.getline( buffer, sizeof(buffer)) ) {
      if( stream.fail() ) {
         cout << "Unerwarteter Lesefehler!" << endl;
         break;
      }
      cout << cnt++ << " : " << buffer << endl;
   }
   cout << endl;
}

// Zeilenweise kopieren
void copyLine4Line( ifstream& istream, ofstream& ostream ) {
   char buffer[512];
   while( istream.getline( buffer, sizeof(buffer)) ) {
      if( istream.fail() ) {
         cout << "Unerwarteter Lesefehler!" << endl;
         break;
      }
      ostream << buffer << endl;
      if( ostream.fail() ) {
         cout << "Unerwarteter Schreibfehler!" << endl;
         break;
      }
   }
}
```

Wollen Sie das Ganze nicht mit C-Strings, sondern mit der Standardklasse `string` realisieren, können Sie auch die globale Funktion `getline()` verwenden. Diese hat als ersten Parameter den Typ `ifstream` und arbeitet auch mit der Klasse `string`. Umgeschrieben würde die Funktion `lineOutput` folgendermaßen aussehen:

```
// fstream6.cpp
#include <iostream>
#include <fstream>
using namespace std;

void lineOutput( ifstream& stream );

int main( int argc, char **argv ) {
   // Datei zum Lesen öffnen
   ifstream f1("schiller.txt");
   if( f1.is_open() != true ) {
      cerr << "Konnte Datei schiller.txt nicht öffnen!"
           << endl;
      //return 1;
   }
   // Zeichenweise einlesen und ausgeben
   lineOutput( f1 );
   f1.close();
   return 0;
}

// Zeilenweise auslesen mit der globalen Funktion getline()
void lineOutput ( ifstream& stream ) {
   string buffer;
   int cnt=1;
   while( getline( stream, buffer ) ) {
      cout << cnt++ << " : " << buffer << endl;
   }
   cout << endl;
}
```

Blockweise Ein-/Ausgabe

Wollen Sie eine bestimmte Anzahl von Bytes (bzw. Blöcken) auf einmal lesen bzw. schreiben, dann können Sie die Methoden `read()` und `write()` verwenden, die beide unformatiert eine bestimmte Anzahl von Bytes lesen bzw. schreiben können. Das folgende Beispiel kopiert eine komplette Datei in eine andere Datei in nur einem Lese- und Schreibvorgang. Dazu verwenden wir ein dynamisches `char`-Array:

```
// fstream7.cpp
#include <iostream>
#include <fstream>
using namespace std;

void copyAllTogether( ifstream& istream, ofstream& ostream );

int main( int argc, char **argv ) {
   // Datei zum Lesen öffnen
   ifstream f1("schiller2.txt");
   ofstream f2("backSchiller.txt");
   if( f1.is_open() != true ) {
      cerr << "Konnte Datei schiller.txt nicht öffnen!"
           << endl;
      //return 1;
   }
   if( f2.is_open() != true ) {
      cerr << "Konnte Datei backSchiller.txt nicht öffnen!"
           << endl;
      //return 1;
   }
   // Blockweise kopieren
   copyAllTogether( f1, f2 );
   f1.close();
   f2.close();
   return 0;
}

// Blockweise kopieren
void copyAllTogether(ifstream& istream,ofstream& ostream) {
   char *buffer;
   size_t size;
   streamoff curpos = istream.tellg();
   istream.seekg(0, ios::end);
   size = istream.tellg();
   istream.seekg(curpos);
   // Speicher reservieren
   buffer = new char[size];
   istream.read( buffer, size );
   ostream.write( buffer, size );
   // Testweise auf die Standardausgabe
   cout << buffer << endl;
   ostream.seekp(size);
   ostream.put('\0');
   // Speicher wieder freigeben
```

```
   delete buffer;
}
```

Das Programm bei der Ausführung:

```
Alles wiederholt sich nur im Leben,
Ewig jung ist nur die Phantasie,
Was sich nie und nirgends hat begeben,
Das allein veraltet nie!
```

Dieser Inhalt sollte sich jetzt auch in der Datei *backSchiller.txt* befinden. Im Beispiel wurden bereits die Methoden `seekg()`, `seekp()` und `tellg()` für den wahlfreien Zugriff verwendet, die im nächsten Abschnitt näher erläutert werden.

Wahlfreier Dateizugriff

Mit dem wahlfreien Zugriff haben Sie die Möglichkeit, an einer gewünschten Position in einer Datei zuzugreifen, sowohl lesend als auch schreibend. Dazu sind Methoden vorhanden, mit denen Sie die aktuelle Lese- bzw. Schreibposition verändern können. Zu diesem Zweck sind in den Klassen `istream` und `ostream` `seek`-Funktionen definiert. Hierzu die Methoden im Überblick:

Methode	Beschreibung
`istream& seekg( long pos );`	Setzt den `get`-Zeiger auf den Byte-Offset `pos` relativ zum Dateianfang.
`istream& seekg ( long pos, ios::seekdir f );`	Setzt den `get`-Zeiger auf den Byte-Offset `pos` relativ zur im zweiten Argument angegebenen Position.
`long tellg();`	Gibt den aktuellen Byte-Offset des `get`-Zeigers relativ zum Dateianfang zurück.
`ostream& seekp( long pos );`	Setzt den `put`-Zeiger auf den Byte-Offset `pos` relativ zum Dateianfang.
`ostream& seekp( long pos, ios::seekdir f );`	Setzt den `put`-Zeiger auf den Byte-Offset `pos` relativ zur im zweiten Argument angegebenen Position.
`long tellp();`	Gibt den aktuellen Byte-Offset des `put`-Zeigers relativ zum Dateianfang zurück.

Tabelle 7.41 Methoden zur Positionierung von »istream« und »ostream«

Für das zweite Argument von `seekg()` bzw. `seekp()` können Sie folgende Positionierungs-Flags vom Typ `ios::seekdir` verwenden:

Flag	Beschreibung
ios::beg	relativ zum Dateianfang positionieren
ios::cur	relativ zur aktuellen Position positionieren
ios::end	relativ zum Dateiende positionieren

Tabelle 7.42 Flags zur Positionierung

Im Listing *fstream7.cpp* haben Sie bereits gesehen, wie man diese Methoden in der Praxis verwenden kann. Hierzu nochmals die Funktion `copyAllTogether` mit einer anschließenden Erläuterung:

```
// Blockweise kopieren
void copyAllTogether(ifstream& istream,ofstream& ostream) {
   char *buffer;
   size_t size;
   streamoff curpos = istream.tellg();
   istream.seekg(0, ios::end);
   size = istream.tellg();
   istream.seekg(curpos);
   // Speicher reservieren
   buffer = new char[size];
   istream.read( buffer, size );
   ostream.write( buffer, size );
   // Testweise auf die Standardausgabe
   cout << buffer << endl;
   ostream.seekp(size);
   ostream.put('\0');
   // Speicher wieder freigeben
   delete buffer;
}
```

Am Anfang dieser Funktion haben wir mit `tellg()` die aktuelle Position des `get`-Zeigers (oder auch Lesezeigers) in `curpos` gesichert. Anschließend positionieren wir den Lesezeiger mit `seekg()` auf das Ende der Datei, um die Größe der Datei in Bytes zu ermitteln. Weil wir den Lesezeiger wieder benötigen, setzen wir diesen mit `seekg()`auf die gesicherte Position – hier den Anfang. Hierbei hätte man genauso gut das Flag `ios::beg` verwenden können. Da jetzt die Dateigröße bekannt ist, kann man entsprechend Speicher reservieren, um die komplette Datei auf einmal unformatiert in einen Puffer einzulesen, um diesen Puffer anschließend in eine andere Datei zu schreiben.

Da beim unformatierten Schreiben mit `write()` nicht sichergestellt ist, dass das Stringende-Zeichen mit kopiert wurde, wird auch noch der `put`-Zeiger mit `seekp()` ans Ende der Datei gesetzt und dann das Stringende-Zeichen eingefügt.

Fehlerbehandlung

Natürlich können auch bei der Arbeit mit Dateien diverse Fehler auftreten. Beispielsweise kann eine Datei gar nicht existieren, oder beim Lesen bzw. Schreiben tritt ein Fehler auf. Hier können Sie wieder auf die bereits bekannten Methoden zurückgreifen. Die wichtigsten sollen hier nochmals zusammengefasst werden.

good()

Ob alles funktioniert hat, können Sie mit der Methode `good()` abfragen. Gleichwertig dazu kann man auch einfach nur das Stream-Objekt abfragen. Beides liefert bei sachgemäßem Ablauf `true`, ansonsten `false` zurück:

```
fstream file;
...
if (file.good()) {
    // File-Stream ist in Ordnung
}
...
// Gleichwertig wie file.good() ...
if ( file ) {
    // File-Stream ist in Ordnung
}
```

Dateiende – eof()

Erreicht der File-Stream das Ende der Datei, wird `EOF` (*End of File*) zurückgegeben. Um das Dateiende von anderen Fehlern zu unterscheiden, müssen Sie die Methode `eof()` verwenden. Gibt diese Methode `true` zurück, wurde das Dateiende erreicht, ansonsten wird `false` zurückgegeben. Zum Beispiel lesen Sie folgendermaßen

```
while( !f1.eof()) {
   f1.get(ch);
   cout.put(ch);
}
```

zeichenweise aus einer Datei ein, bis das Ende der Datei (`EOF`) erreicht wurde.

Fehler löschen

Wenn ein Fehler für einen File-Stream ausgegeben wurde, müssen Sie diesen mit der Methode `clear()` wieder löschen, sofern Sie den File-Stream weiterverwenden wollen, da sonst das Fehler-Bit gesetzt bleibt.

»fail()« und »bad()«

Tritt ein Fehler beim Schreiben oder beim Lesen auf oder konnte ein Aufruf nicht komplett ausgeführt werden, können Sie die Methode `fail()` abfragen. `fail()` gibt `false` zurück, wenn alles in Ordnung ist, ansonsten wird `true` zurückgegeben:

```
while( !f1.eof()) {
   f1.get(ch);
   if( f1.fail() ) {
      cerr << "Fehler beim Lesen ...!" << endl;
   }
   cout.put(ch);
}
```

Schwerwiegende Fehler kann man mit der Methode `bad()` abfragen. Ein solcher Fehler führt dazu, dass mit dem File-Stream keine weiteren Operationen mehr möglich sind.

Exceptions

Natürlich können Sie auch die Exceptions zur Fehlerbehandlung verwenden (siehe Kapitel 6, »Exception-Handling«). Als Parameter für die Methode `exceptions()` von File-Streams können Sie die Konstanten `ios::failbit`, `ios::badbit` und `ios::eofbit` verwenden.

7.2.3 Klassen für String-Streams

Die Ein-/Ausgabeoperatoren << und >> haben nicht nur die Aufgabe, Dateien zu lesen bzw. zu schreiben, sondern sie konvertieren auch die Daten von der Binär-Darstellung in die für uns lesbare ASCII-Form und umgekehrt. Diese Möglichkeit der formatierten Ein-/Ausgabe für Dateien benötigt man manchmal auch für Strings. Für solche Zwecke stehen String-Streams zur Verfügung. String-Streams sind ein komfortables Mittel, mit dem Sie unterschiedliche Datenformate ineinander konvertieren können. Das einfachste Beispiel einer solchen Konvertierung ist die Umwandlung eines Strings in einen numerischen Wert. Aber für String-Streams gibt es eine Menge weiterer wichtiger Anwendungsgebiete:

- Sie wollen Datensätze in einen bestimmten String umwandeln. Dies ist häufig der Fall bei Programmen, die Daten mit anderen Programmen austauschen, wie zum Beispiel bei einem Netzwerk. Häufig werden die Daten vom Sender mit Zusatzinformationen versehen, die beim Empfänger wieder entfernt werden.
- Bei Programmen mit einer grafischen Oberfläche stehen Ihnen im Allgemeinen nur Funktionen zur Verfügung, mit denen Sie einen gewöhnlichen String

ausgeben können. Wenn Sie hierbei allerdings binäre Daten haben, müssen Sie diese zunächst in einen String umwandeln.

- Bei der Eingabe über die Tastatur wird eine Eingabe in Form eines Strings übernommen. Erst anschließend wird diese Eingabe ins richtige Format konvertiert und umgewandelt.

In Abbildung 7.2 können Sie erkennen, dass Ihnen für die String-Streams die Klassen `istringstream`, `ostringstream` und `stringstream` zur Verfügung stehen. Alle Klassen sind in der Header-Datei <*sstream*> deklariert.

Die bisher vorgestellten Streams stehen im Zusammenhang mit den Ein-/Ausgabe-Streams `cin`, `cout` und `cerr`. Dann gab es auch noch die File-Streams, die Dateien zur Ein-/Ausgabe verwenden. Die jetzt hier vorgestellten String-Streams nutzen den Typ `string` zur Ein-/Ausgabe.

[»] **Hinweis**

Die erwähnten String-Streams aus <sstream> sind nicht zu verwechseln mit den »veralteten« String-Streams der Klassen `istrstream`, `ostrstream` und `strstream` aus <strstream>, die noch mit dem `char*`-Typ arbeiten. Von diesen muss abgeraten werden, da es noch mögliche Speicherlecks geben kann. Diese Klasse ist nur noch aus Kompatibilitätsgründen vorhanden.

Alle drei Klassen, `istringstream`, `ostringstream` und `stringstream`, besitzen je zwei Konstruktoren, in denen jeweils ein Puffer vom Typ `stringbuf` angelegt wird. Diese Klasse wiederum ist eine `public`-Vererbung der Klasse `streambuf`. Hier ein Überblick über die Syntax dieser Konstruktoren:

Konstruktor	Beschreibung
`explicit istringstream ( openmode mode = in );`	Erzeugt einen String-Stream im angegebenen Modus. Standardmäßig wird aus diesem String-Stream gelesen.
`explicit istringstream ( const string & str, openmode mode = in );`	Erzeugt einen String-Stream im angegebenen Modus und initialisiert diesen mit `str`. Standardmäßig wird aus diesem String-Stream gelesen.
`explicit ostringstream ( openmode mode=out );`	Erzeugt einen String-Stream im angegebenen Modus. Standardmäßig wird in diesem String-Stream geschrieben.
`explicit ostringstream ( const string & str, openmode mode = out );`	Erzeugt einen String-Stream im angegebenen Modus und initialisiert diesen mit `str`. Standardmäßig wird in diesem String-Stream geschrieben.

Tabelle 7.43 Konstruktoren der Klassen »istringstream«, »ostringstream« und »stringstream«

Konstruktor	Beschreibung
`explicit stringstream ( openmode mode=in|out);`	Erzeugt einen String-Stream im angegebenen Modus. Standardmäßig kann dieser String-Stream gelesen und beschrieben werden.
`explicit stringstream ( const string & str, openmode mode=in|out);`	Erzeugt einen String-Stream im angegebenen Modus und initialisiert diesen mit `str`. Standardmäßig kann dieser String-Stream gelesen und beschrieben werden.

Tabelle 7.43 Konstruktoren der Klassen »istringstream«, »ostringstream« und »stringstream« (Forts.)

Der Modus, den Sie hier für `openmode` verwenden können, ist derselbe wie schon bei den File-Streams. Hier nochmals der Überblick dazu:

Flag	Beschreibung
`ios::in`	String-Stream wird zum Lesen verwendet.
`ios::out`	String-Stream wird zum Schreiben verwendet.
`ios::app`	Vor jeder Schreiboperation wird der Zeiger auf das Stringende gesetzt.
`ios::trunc`	Beim Anlegen eines String-Streams wird die Länge auf `0` gesetzt. Achtung, löscht den Inhalt des Strings!
`ios::ate`	String-Stream öffnen und den Zeiger auf das Streamende setzen
`ios::binary`	Lese- und Schreiboperationen im Binärmodus ausführen

Tabelle 7.44 Öffnungsmodus der Klasse »ios«

Ein Gräuel für jeden Programmierer ist die Umwandlung numerischer Werte in Strings und umgekehrt. In C wurden hierfür gerne die Funktionen `sprintf()` und `sscanf()` genutzt. Dabei musste man immer auf unleserliche Formatierungsangaben zurückgreifen, und `sscanf()` war zudem noch anfällig für einen Pufferüberlauf (Buffer Overflow), was häufig mit einem Absturz des Programms verbunden war. Mit den String-Streams gehört so etwas der Vergangenheit an.

Hierzu ein Beispiel, das zeigt, wie Sie Daten in Strings mit `ostringstream` konvertieren können:

```
// sstream1.cpp
#include <iostream>
#include <sstream>
using namespace std;

int main( int argc, char **argv ) {
   ostringstream ostr;
   int ival = 123;
```

```
   float fval = 123.123;
   char cval = 'X';
   string zielStr;
   // Diese Daten mit << in den String-Stream ostr
   // hineinschieben und die Binärdarstellung in
   // eine ASCII-Darstellung konvertieren
    ostr << "ival = " << ival << endl;
    ostr << "fval = " << fval << endl;
    ostr << "cval = " << cval << endl;
    // Diese Daten werden mit der Methode str() herausgeholt
    zielStr = ostr.str();
    // Hier die ASCII-Darstellung
    cout << zielStr << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
ival = 123
fval = 123.123
cval = X
```

Neu in diesem Beispiel ist die Methode `str()`, mit der die komplette Zeichenfolge des String-Streams in den String `zielStr` kopiert wird. Diese Methode gibt es in zwei Versionen:

Methode	Beschreibung
`string` **`str`**`() const;`	Kopiert die Zeichenfolge vom String-Stream-Puffer in einen String und gibt diesen zurück. Sind im Stream-Puffer keine Zeichen, wird ein leerer String zurückgegeben.
`void` **`str`**`(const string& str);`	Kopiert den String `str` in den String-Stream-Puffer. Beachten Sie allerdings, dass der alte Inhalt verloren geht.

Tabelle 7.45 Methoden der Klassen »istringstream«, »ostringstream« und »stringstream«

Häufig will man aber auch einen String in einen numerischen Wert konvertieren. Nichts ist leichter als das. Man verwendet einfach die Klasse `stringstream`, weil es damit möglich ist, aus dem Stream zu lesen und zu schreiben. Ein einfaches Beispiel dazu:

```
// sstream2.cpp
#include <iostream>
#include <sstream>
using namespace std;
```

```
int main( int argc, char **argv ) {
   stringstream strstr1;
   stringstream strstr2;
   stringstream strstr3;
   int ival;
   float fval;
   string strival="100";
   string strfval="123.123";

   // String2Int
   strstr1 << strival;
   strstr1 >> ival;

   //String2Float
   strstr2 << strfval;
   strstr2 >> fval;

   // Der Beweis 1
   cout << ival + ival << endl;
   // Der Beweis 2
   cout << fval + fval << endl;

   // Und wieder einen String daraus machen
   strstr3 << ival << " und " << fval << endl;
   string ifval = strstr3.str();
   cout << ifval;
   return 0;
}
```

Das Programm bei der Ausführung:

```
200
246.246
100 und 123.123
```

Natürlich lässt sich das Ganze auch nur mit `istringstream` realisieren, womit die Daten aus einem String in den Stream gelesen werden. Ein einfaches Beispiel hierzu:

```
// sstream3.cpp
#include <iostream>
#include <sstream>
using namespace std;

int main( int argc, char **argv ) {
   istringstream istr;
```

```
   string quellString = "123 123.123 X";
   int ival;
   float fval;
   char cval;

    // Daten als String in istr schreiben:
    istr.str(quellString);
    // Diese Daten mit >> aus dem String-Stream istr
    // herausschieben und die ASCII-Darstellung in
    // eine Binär-Darstellung konvertieren
    istr >> ival;
    istr >> fval;
    istr >> cval;
    cout << ival << " " << fval << " " << cval << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
123 123.123 X
```

Sicherlich stellen Sie sich jetzt die Frage, was passiert, wenn man versucht, eine Zeichenkette wie »123.123« in einen `int` zu konvertieren. Dass dies ein Fehler ist, leuchtet ein; abfragen kann man diesen mit der Methode `eof()`. Wenn der String-Stream nach einer Konvertierung auf `eof()` geprüft `true` zurückgibt, hat alles funktioniert. Trat ein Fehler auf, gibt `eof()` `false` zurück, und die Konvertierung ist fehlgeschlagen. Hierzu ein Beispiel:

```
// sstream4.cpp
#include <iostream>
#include <sstream>
using namespace std;

int main( int argc, char **argv ) {
   istringstream istr;
   string quellString = "123.123";
   int ival;
   // Daten als String in istr schreiben:
   istr.str(quellString);
   istr >> ival;
   if( istr.eof() ) {
      cout << "Erfolgreich konvertiert..." << endl;
   }
   else {
      cout << "Fehler beim Konvertieren ..." << endl;
   }
```

```
   return 0;
}
```

Das Programm bei der Ausführung:

```
Fehler beim Konvertieren ...
```

Die hier vorgestellten String-Streams schreien geradezu nach Templates. Damit können Sie eine recht universelle Schablone erstellen, mit der Sie Konvertierungen beliebiger Basisdatentypen durchführen können. Hierzu folgt nun ein einfaches Beispiel, wie ein solches Template aussehen kann. In der Praxis empfiehlt es sich, außerdem noch den Fehler mit einer Exception (siehe Kapitel 6, »Exception-Handling«) abzufangen.

```
// sstream5.cpp
#include <iostream>
#include <cstdlib>
#include <sstream>
using namespace std;

template <class T1, class T2>
void typconverter( const T1& input, T2& output) ;

int main( int argc, char **argv ) {
   string fstring = "123.123";
   string istring = "100";
   string teststring1, teststring2;
   int ival;
   float fval;

   // string2float
   typconverter( fstring, fval );
   cout << fval << endl;
   // string2int
   typconverter( istring, ival );
   cout << ival << endl;
   // float2string
   typconverter( fval, teststring1 );
   cout << teststring1 << endl;
   // int2string
   typconverter( ival, teststring2 );
   cout << teststring2 << endl;
   return 0;
}

// Erster Parameter: die Quelle
```

```
// Zweiter Parameter: das Ziel
template <class T1, class T2>
void typconverter( const T1& input, T2& output) {
    stringstream ss;
    ss << input;
    ss >> output;
    if (! ss.eof()) {
       cout << "Konvertierung fehlgeschlagen ..." << endl;
       exit(1); // ... besser wäre eine Exception
    }
}
```

Das Programm bei der Ausführung:

```
123.123
100
123.123
100
```

Auf den Stream-Puffer zugreifen

Wenn Sie die Methode `rdbuf()` aufrufen, liefert diese die Pufferadresse des Streams zurück. Dadurch können Sie mit Methoden der Klasse `streambuf` (siehe nächsten Abschnitt) direkt darauf zugreifen.

7.2.4 Die Klasse »streambuf«

`streambuf` ist eine abstrakte Basisklasse, die die grundlegenden Eigenschaften und Methoden von Stream-Puffern zur Verfügung stellt. Jeder Stream (String-Stream, Datei-Stream und Standard-I/O-Stream) arbeitet mit Stream-Puffern, meistens mit einer von `streambuf` abgeleiteten Klasse, beispielsweise `filebuf`, `stdiobuf` (siehe auch Abbildung 7.2).

Die Klasse `streambuf` verwendet Puffer, in denen Zeichen eingefügt (`put`) oder aus denen Zeichen entnommen (`get`) werden können. Das Einfügen bzw. Entnehmen wird mit einem oder zwei Zeigern (einem `get`- und/oder einem `put`-Zeiger) realisiert, die die Position angeben, an der ein Zeichen eingefügt bzw. entnommen werden kann.

[»] **Hinweis**

Aus Platzgründen sei hier nur kurz auf die einzelnen Methoden der Klasse `streambuf` eingegangen. Wenn Sie weitergehende Informationen erhalten möchten, sollten Sie die Standardreferenz Ihres Compilers zu Rate ziehen.

Zur Verwaltung eines Stream-Puffers werden drei Zeiger verwendet, die alle als `private` deklariert sind:

- Zeiger auf den Anfang des Puffers
- Zeiger auf die nächste zu beschreibende bzw. zu lesende Position im Puffer
- Zeiger auf das Ende des Puffers

Puffer anlegen und positionieren

Methode	Beschreibung
`streambuf* pubsetbuf( char* s, int n );`	Ruft die virtuelle Methode `setbuf()` zum Anlegen eines Puffers auf. Wie diese Methode implementiert ist, wird nicht vom Standard definiert. Bei der Klasse `filebuf` wird das `char`-Array `s` mit der Länge `n` als Puffer verwendet. Ein Aufruf von `pubsetbuf(0,0)` würde eine ungepufferte Datenübertragung bedeuten. Der Rückgabewert ist entweder der Zeiger `*this` oder `0`.
`long pubseekoff( long off, ios::seekdir w, ios::openmode which = ios::in | ios::out);`	Setzt den Schreib- bzw. Lesezeiger auf die Position `off`, relativ zur Position `w`. Zurückgegeben wird die neue Schreib-/Leseposition oder im Fall eines Fehlers `–1`.
`int pubseekpos( long sp, ios::openmode which = ios::in | ios::out);`	Setzt den Schreib- bzw. Lesezeiger auf die absolute Position `sp`. Zurückgegeben wird die neue Schreib-/Leseposition oder im Fall eines Fehlers `–1`.
`int pubsync();`	Überträgt alle sich noch im Puffer befindlichen Zeichen und leert somit diesen Puffer. Zurückgegeben wird ordnungsgemäß `0` oder im Fall eines Fehlers `–1`.

Tabelle 7.46 Methoden zum Anlegen und Positionieren eines Puffers

Zeichen lesen

Methode	Beschreibung
`int in_avail();`	Gibt die Anzahl der noch nicht gelesenen Zeichen im Puffer zurück oder `–1`, wenn sich keine Zeichen im Puffer befinden.
`int sbumpc();`	Gibt das nächste Zeichen im Puffer zurück und erhöht den `next`-Zeiger. Ist kein Zeichen verfügbar, wird `traits::eof()` zurückgegeben.
`int snextc();`	Ruft die Methode `sbumbc()` auf. Ist kein Zeichen verfügbar, wird `traits::eof()` zurückgegeben. Ist ein Zeichen verfügbar, wird `sgetc()` aufgerufen.

Tabelle 7.47 Methoden zum Lesen von Zeichen aus dem Puffer

Methode	Beschreibung
`int sgetc();`	Gibt das nächste Zeichen im Puffer zurück, ohne den `next`-Zeiger zu inkrementieren. Ist kein Zeichen verfügbar, wird `traits::eof()` zurückgegeben.
`int sgetn( char* s, int n );`	Liest `n` Zeichen aus dem Puffer und schreibt diese in das mit `s` adressierte Array. Sind keine Zeichen mehr vorhanden, wird das Einlesen abgebrochen. Zurückgegeben wird die Anzahl der gelesenen Zeichen.

Tabelle 7.47 Methoden zum Lesen von Zeichen aus dem Puffer (Forts.)

Zeichen zurückstellen und schreiben

Methode	Beschreibung
`int sputc( char ch );`	Schreibt das Zeichen `ch` in die aktuelle Schreibposition des Puffers und erhöht den `put`-Zeiger. Der Rückgabewert ist das Zeichen `ch` oder bei einem Fehler `traits::eof()`.
`int sputn( const char* s, int n );`	Schreibt `n` Zeichen aus dem Array `s` in den Puffer. Ist der Puffer voll, wird das Schreiben abgebrochen. Zurückgegeben wird die Anzahl der geschriebenen Zeichen.
`int sputbackc( char ch );`	Stellt das zuletzt gelesene Zeichen `ch` in den Puffer zurück und reduziert den `next`-Zeiger um 1. Der Rückgabewert ist das Zeichen `ch` oder bei einem Fehler `traits::eof()`.
`int sungetc();`	Dekrementiert den `next`-Zeiger und gibt das zuletzt gelesene Zeichen oder im Fall eines Fehlers `traits::eof` zurück.

Tabelle 7.48 Methoden zum Schreiben und Zurückstellen von Zeichen

Das folgende Beispiel soll zeigen, wie man aus einer geöffneten Datei blockweise Zeichen einlesen kann, bis das Dateiende erreicht ist. Für jeden eingelesenen Block wird die Größe des Puffers ausgegeben, der noch nicht eingelesen wurde, das heißt die Zeichen, die sich noch im Puffer befinden.

```
// streambuf1.cpp
#include <iostream>
#include <fstream>
using namespace std;

int main( int argc, char **argv ) {
   // Blockgröße 10 Zeichen
   const int N = 10;
   int count;
```

```
   char text_b[N];
   fstream filestream("schiller.txt");
   streambuf* pbuf = filestream.rdbuf();

   cout << "Noch " << pbuf->in_avail()
        << " Zeichen zum Lesen ..." << endl;

   /* Bereich get vor dem sgetn-Aufruf prüfen */
   while ( (count = pbuf->sgetn(&text_b[0], N)) > 0) {
      text_b[count]='\0';
      cout << text_b << endl;
      cout << "Noch " << pbuf->in_avail()
           << " Zeichen zum Lesen ..." << endl;
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Noch 98 Zeichen zum Lesen ...
Ewig jung
Noch 86 Zeichen zum Lesen ...
ist nur di
Noch 76 Zeichen zum Lesen ...
e Phantasi
Noch 66 Zeichen zum Lesen ...
e,
Was sic
Noch 56 Zeichen zum Lesen ...
h nie und
Noch 46 Zeichen zum Lesen ...
nirgends h
Noch 36 Zeichen zum Lesen ...
at begeben
Noch 26 Zeichen zum Lesen ...
,
Das alle
Noch 16 Zeichen zum Lesen ...
in veralte
Noch 6 Zeichen zum Lesen ...
t nie!
Noch 0 Zeichen zum Lesen ...
```

7.2.5 Die Klasse »filebuf«

Die Klasse `filebuf` entsteht durch eine `public`-Vererbung der Klasse `streambuf`. Allerdings werden im Gegensatz zu `streambuf` Dateien als Quelle oder Ziel für die Zeichenübertragung verwendet. Durch das Schreiben in die Datei werden die einzelnen Zeichen im Puffer weniger und durch das Lesen wiederum mehr. Eben durch diese `public`-Vererbung stehen für die Klasse `filebuf` alle öffentlichen Methoden der Klasse `streambuf` bereit. Neben einem Default-Konstruktor `filebuf()`, der einen Stream-Puffer anlegt, ohne eine Datei zu öffnen, und dem Destruktor `~filebuf()` stehen Ihnen in der Klasse `filebuf` noch die folgenden Methoden zur Verfügung:

Methode	Beschreibung
`bool is_open() const;`	Existiert eine entsprechende Datei und ist diese geöffnet, wird `true`, ansonsten `false` zurückgegeben.
`filebuf* open( const char* s, ios::openmode which );`	Öffnet die Datei `s` im angegebenen Modus `which`, legt einen Stream-Puffer an und setzt den `next`-Zeiger auf eine entsprechende Position (abhängig vom Öffnungsmodus). Im Fall eines Fehlers wird `NULL` zurückgegeben – ansonsten ein Zeiger auf `*this`.
`filebuf* close();`	Wenn die Datei geöffnet wurde, wird diese hiermit geschlossen. Auch hier wird im Fall eines Fehlers `NULL` zurückgegeben – ansonsten der Zeiger `*this`.

Tabelle 7.49 Methoden zum Öffnen und Schließen von Dateien

Hierzu ein einfaches Beispiel, das die öffentlichen Methoden von `filebuf` in der Praxis demonstriert und blockweise eine Datei kopiert (eine Erweiterung des Listings *streambuf1.cpp*):

```
// filebuf1.cpp
#include <iostream>
#include <fstream>
using namespace std;

int main( int argc, char **argv ) {
   const int N = 10;
   int count;
   char text_b[N];
   filebuf pbuf;
   filebuf zbuf;

   pbuf.open("schiller.txt", ios::in);
   zbuf.open("backSchiller.txt", ios::out);
   if( !pbuf.is_open()) {
```

```
      cout << "Konnte schiller.txt nicht öffnen"
           << endl;
      return 1;
   }
   if( !zbuf.is_open()) {
      cout << "Konnte backSchiller.txt nicht öffnen"
           << endl;
      return 1;
   }
   while ( (count = pbuf.sgetn(&text_b[0], N)) > 0) {
      text_b[count]='\0';
      zbuf.sputn( text_b, count );
   }
   pbuf.close();
   zbuf.close();
   return 0;
}
```

7.2.6 Die Klasse »stringbuf«

Die Klasse `stringbuf` entsteht ebenfalls aus einer `public`-Vererbung der Klasse `streambuf`. Damit werden Methoden der Basisklasse zum Anlegen eines Puffers für String-Streams realisiert. Diese Puffer werden vorwiegend zum Positionieren des `next`-Zeigers und zur Verwaltung von Pufferunterläufen verwendet. Durch die `public`-Vererbung stehen sämtliche öffentlichen Methoden der Klasse `streambuf` auch für `stringbuf` zur Verfügung. Im `public`-Teil der Klasse `stringbuf` sind noch zusätzlich folgende Methoden definiert:

Methode	Beschreibung
`explicit stringbuf ( ios_base::openmode which = ios_base::in|ios_base::out );`	Konstruktor; legt einen Stream-Puffer im angegebenen Modus an.
`explicit stringbuf ( const string& str, ios_base::openmode which = ios_base::in|ios_base::out );`	Konstruktor; legt einen Stream-Puffer im angegebenen Modus an und kopiert den String `str` in den Puffer.
`string str() const;`	Kopiert die Zeichenfolge vom String-Stream-Puffer in einen String und gibt diesen zurück. Sind im Stream-Puffer keine Zeichen, wird ein leerer String zurückgegeben.
`void str( const string& str );`	Kopiert den String `str` in den String-Stream-Puffer. Beachten Sie allerdings, dass der alte Inhalt verloren geht.

Tabelle 7.50 Methoden der Klasse »stringbuf«

Ein einfaches Beispiel hierzu:

```
// stringbuf1.cpp
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main( int argc, char **argv ) {
   stringbuf stringbuf1("Test-Puffer",
                        ios::in|ios::out|ios::app );
   stringbuf stringbuf2;
   string meinString1;
   string meinString2;

   stringbuf1.sputn(" - kommt hinten hin", 19 );
   stringbuf2.sputn("Noch ein Beispiel", 17 );

   meinString1=stringbuf1.str();
   meinString2=stringbuf2.str();

   cout << meinString1 << endl;
   cout << meinString2 << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Test-Puffer - kommt hinten hin
Noch ein Beispiel
```

7.3 Numerische Bibliothek(en)

Die Standardbibliothek von C++ (und C) bietet eine Menge interessanter Klassen und Bibliotheken, die für komplexe mathematische Berechnungen verwendet werden können. Natürlich kann auch hierbei nicht auf jede Einzelheit eingegangen werden, so dass der Abschnitt nur für Referenzzwecke geeignet ist.

7.3.1 Komplexe Zahlen (»complex«-Klasse)

In der numerischen Bibliothek ist das Klassen-Template `complex<T>` zur Darstellung komplexer Zahlen in kartesischen Koordinaten definiert. Dabei sind neben den verschiedenen Konstruktoren auch Methoden für den Real- und Imaginärteil einer komplexen Zahl und die Zuweisungsoperatoren definiert. Die arithmeti-

schen Operatoren und Vergleichsoperatoren – wobei hier nur Prüfungen auf Gleichheit (==) und Ungleichheit (!=) Sinn machen – sind alle global deklariert, ebenso die Ein-/Ausgabeoperatoren << und >> zum Einlesen bzw. Ausgeben komplexer Zahlen.

Hinweis [«]

Das kartesische Koordinatensystem ist nach Descartes benannt. Es handelt sich um das am häufigsten verwendete Koordinatensystem, da sich geometrische Sachverhalte darin am besten beschreiben lassen.

Die komplexen Zahlen werden in der Header-Datei <*complex*> durch spezialisierte Templates für die Typen `float`, `double` und `long double` realisiert. Damit kann genauso gerechnet werden wie mit reellen Zahlen.

In der folgenden Tabelle (7.51) werden alle Funktionen aufgelistet, die mit `complex`-Zahlen möglich sind. `T` steht hierbei für einen der Typen `float`, `double` oder `long double`.

Funktion	Beschreibung
`T real(const complex<T>& x);`	Gibt den Realteil von `x` zurück.
`T imag(const complex<T>& x);`	Gibt den Imaginärteil von `x` zurück.
`T abs(const complex<T>& x);`	Gibt den absoluten Betrag von `x` zurück.
`T arg(const complex<T>& x);`	Gibt den Phasenwinkel von `x` in rad zurück.
`T norm(const complex<T>& x);`	Gibt den quadrierten Betrag von `x` zurück.
`complex<T> conj(` `  const complex<T>& x);`	Gibt die zu `x` konjugiert-komplexe Zahl zurück.

`complex<T> polar(`
`  const T& rho,`
`  const T& theta);`
	Gibt aus dem Betrag `rho` und der Phase `theta` die entsprechende komplexe Zahl zurück.

`complex<T> cos(` `  const complex<T>& x);`	Gibt den Cosinus von `x` zurück.
`complex<T> cosh(` `  const complex<T>& x);`	Gibt den Cosinus hyperbolicus von `x` zurück.
`complex<T> exp(` `  const complex<T>& x);`	Gibt den Wert der natürlichen Exponentialfunktion an der Stelle `x` zurück.
`complex<T> log(` `  const complex<T>& x);`	Gibt den natürlichen Logarithmus von `x` für den Bereich [-i*p, +i*p] zurück.
`complex<T> log10(` `  const complex<T>& x);`	Gibt den Logarithmus von `x` zur Basis 10 zurück.

Tabelle 7.51 Mathematische Funktionen für »complex«-Zahlen

Funktion	Beschreibung
`complex<T> pow( const complex<T>& x, int y);`	Gibt den Wert von `x` hoch `y` zurück. Der Wert ist durch `exp(y*log(x))` definiert.
`complex<T> pow( const complex<T>& x, const T& y);`	
`complex<T> pow( const T& x, const complex<T>& y);`	
`complex<T> pow( const complex<T>& x, const complex<T>& y);`	
`complex<T> sin( const complex<T>& x);`	Gibt den Sinus von `x` zurück.
`complex<T> sinh( const complex<T>& x);`	Gibt den Sinus hyperbolicus von `x` zurück.
`complex<T> sqrt( const complex<T>& x);`	Berechnet die Quadratwurzel von `x` im Bereich der rechten Halbebene der komplexen Zahlenebene.
`complex<T> tan( const complex<T>& x);`	Gibt den Tangens von `x` zurück.
`complex<T> tanh( const complex<T>& x);`	Gibt den Tangens hyperbolicus von `x` zurück.

Tabelle 7.51 Mathematische Funktionen für »complex«-Zahlen (Forts.)

[»] Hinweis

Bei der Eingabe von komplexen Zahlen sind die Klammern zwingend notwendig – also z. B. (1,1).

Hierzu ein einfaches Beispiel, das zeigt, wie man diese Funktionen verwendet:

```
// complex1.cpp
#include <iostream>
#include <complex>
using namespace std;

int main( int argc, char **argv ) {
   complex <double> a(1,2), b, c;
   cout << "Bitte Wert für b eingeben: ";
   cin >> b;

   c=a+b;
   cout << "Summe   : " << c << " " << abs(c) << endl;
```

```
   c=a/c;
   cout << "Quotient: " << c << " " << abs(c) << endl;
   cout << "Sinus   : " << sin(b) << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Bitte Wert für b eingeben: (2,3)
Summe   : (3,5) 5.83095
Quotient: (0.382353,0.0294118) 0.383482
Sinus   : (9.1545,-4.16891)
```

7.3.2 valarray

In der Header-Datei <*valarray*> befindet sich die Template-Klasse `valarray`, eine ähnliche Klasse wie Sie sie bereits in Abschnitt 5.3.5, »Container«, mit dem Container `vector` kennengelernt haben. Der Unterschied zwischen `valarray` und `vector` besteht darin, dass `valarray` für mathematische Vektoroperationen optimiert wurde. Zwar sind die Methoden dieser Klasse umständlicher, aber dadurch ist es möglich, dass die Compiler-Hersteller diese Klasse einfacher optimieren können.

In der Header-Datei <*valarray*> sind auch die Klassen `slice` und `gslice` definiert, mit denen Sie einen Ausschnitt aus einem `valarray` abbilden können. Dies wird zum Beispiel für Matrix-Berechnungen verwendet.

Konstruktoren

Zum Anlegen eines Objekts der Klasse `valarray` gibt es verschiedene Konstruktoren:

Konstruktor	Beschreibung
`valarray();`	Erzeugt ein `valarray` der Länge `0`.
`explicit valarray(size_t n);`	Erzeugt ein `valarray` der Länge `n`. Die Elemente werden nicht initialisiert.

`valarray(`
`  const T& val,`
`  size_t n);`
	Erzeugt ein `valarray` der Länge `n`, in dem alle Werte mit `val` initialisiert werden.

`valarray(`
`  const T* ptr,`
`  size_t n);`
	Erzeugt ein `valarray` der Länge `n`, in dem alle Elemente mit den ersten `n` Werten initialisiert werden, auf die `ptr` zeigt. Dieser Konstruktor wird gewöhnlich dazu verwendet, um ein C-Array in ein `valarray` umzuwandeln.

Tabelle 7.52 Konstruktoren (und Destruktor) der Klasse »valarray«

Konstruktor	Beschreibung
`valarray(` `  const valarray<T>& v);`	Kopierkonstruktor; erzeugt ein `valarray` als Kopie des `valarray` `n`.
`valarray(` `  const slice_array<T>& v);`	Erzeugt ein `valarray` aus dem Vektor `v` vom Typ `slice_array`.
`valarray(` `  const gslice_array<T>& v);`	Erzeugt ein `valarray` aus dem Vektor `v` vom Typ `gslice_array`.
`valarray(` `  const mask_array<T>& v);`	Erzeugt ein `valarray` aus dem Vektor `v` vom Typ `mask_array`.
`valarray(` `  const indirect_array<T>& v);`	Erzeugt ein `valarray` aus dem Vektor `v` vom Typ `indirect_array`.
`~valarray()`	Destruktor; zerstört ein `valarray`.

Tabelle 7.52 Konstruktoren (und Destruktor) der Klasse »valarray« (Forts.)

[»] **Hinweis**

Die Hilfsklassen `slice_array`, `gslice_array`, `mask_array` und `indirect _array` werden noch extra behandelt.

Hierzu ein einfaches Beispiel, das die Verwendung der Konstruktoren für ein `valarray` demonstriert:

```
// valarray1.cpp
#include <iostream>
#include <valarray>
using namespace std;

int main( int argc, char **argv ) {
   // Ein valarray mit 5 Elementen vom Typ int
   valarray<int> va1(5);
   // Ein valarray mit 5 Elementen vom Typ float,
   // die mit dem Wert 1.1 initialisiert werden
   valarray<float> va2(1.1, 5);

   // Ein C-Array
   int Carr[] = { 1, 2, 4, 8, 16 };
   // Ein valarray aus dem C-Array machen
   valarray<int> va3(Carr, sizeof(Carr)/sizeof(int));
   // Ausgeben ...
   cout << endl << "va1: ";
   for( unsigned int i=0; i < va1.size(); i++ ) {
      cout << va1[i] << " ";
   }
   cout << endl << "va2: ";
```

```
   for( unsigned int i=0; i < va2.size(); i++ ) {
      cout << va2[i] << " ";
   }
   cout << endl << "va3: ";
   for( unsigned int i=0; i < va3.size(); i++ ) {
      cout << va3[i] << " ";
   }
   cout << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
va1: 0 0 0 0 0
va2: 1.1 1.1 1.1 1.1 1.1
va3: 1 2 4 8 16
```

Methoden

Für die Klasse `valarray` gibt es viele Methoden, die in den folgenden Tabellen (7.53 und 7.54) näher erläutert werden. Besonders erwähnt werden soll aber, dass auch bei einem `valarray` der Indexoperator wie üblich verwendet werden kann.

Methode	Beschreibung
`T operator[] (size_t i) const;`	Gibt den Wert des Elements mit dem Index `i` zurück.
`T& operator[] (size_t i);`	Gibt eine Referenz des Elements mit dem Index `i` zurück.
`valarray<T> operator[](slice s) const;`	Gibt einen Teilvektor vom Typ `valarray<T>` zurück, der die Werte der durch `s` indizierten Elemente von `*this` enthält.
`slice_array<T> operator[](slice s);`	Gibt ein Hilfsobjekt vom Typ `slice_array<T>` zurück, das die durch `s` indizierten Elemente von `*this` referenziert.
`valarray<T> operator[]( const gslice& gs) const;`	Gibt einen Teilvektor vom Typ `valarray<T>` zurück, der die Werte der durch `gs` indizierten Elemente von `*this` enthält.
`gslice_array<T> operator[](const gslice& gs);`	Gibt ein Hilfsobjekt vom Typ `gslice_array<T>` zurück, das die durch `gs` indizierten Elemente von `*this` referenziert.
`valarray<T> operator[]( const valarray<bool>& v) const;`	Gibt einen Teilvektor vom Typ `valarray<T>` zurück, der die Elemente von `*this` enthält, für die das Element mit demselben Index in `v` `true` ist.

Tabelle 7.53 Indexoperator

Methode	Beschreibung
`mask_array<T> operator[]( const valarray<bool>& b);`	Gibt ein Hilfsobjekt vom Typ `mask_array<T>` zurück, das die Elemente von `*this` referenziert, für die das Element mit demselben Index in `v` den Wert `true` enthält.
`valarray<T> operator[]( const valarray<size_t>& v) const;`	Gibt einen Vektor vom Typ `valarray<T>` zurück, der aus dem Vektor `*this` durch eine indirekte Indizierung aus `v` entsteht.
`indirect_array<T> operator[]( const valarray<size_t>& v);`	Gibt ein Hilfsobjekt vom Typ `indirect_array<T>` zurück, das die durch `v` indirekt indizierten Elemente von `*this` referenziert.

Tabelle 7.53 Indexoperator (Forts.)

[»]

Hinweis
Die Hilfsklassen `slice_array`, `gslice_array`, `mask_array` und `indirect _array` werden noch extra behandelt.

Methode	Beschreibung
`size_t size() const;`	Gibt die Anzahl der Elemente im `valarray` zurück.
`size_t resize( size_t sz, T c = T() );`	Ändert die Länge des `valarray` auf den Wert `sz`.
`T sum() const;`	Gibt die Summe aller `valarray`-Elemente zurück. Sind keine Elemente vorhanden, ist der Wert undefiniert.
`T min() const;`	Gibt das kleinste `valarray`-Element zurück.
`T max() const;`	Gibt das größte `valarray`-Element zurück.
`valarray<T> shift(int n) const;`	Schiebt jedes `valarray`-Element um `n` Indizes vor, falls `n` positiv ist, oder zurück, wenn `n` negativ ist.
`valarray<T> cshift(int n) const;`	Führt einen Ring-Shift aus. Ist `n` positiv, werden die Elemente um `n` Elemente nach vorn, ist `n` negativ, nach hinten verschoben.
`valarray<T> apply( T func( T ) ) const;`	Bei jedem Element wird die Funktion `func()` mit einem Element als Argument aufgerufen.
`valarray<T> apply( T func( const T&)) const;`	Dito, nur mit einem konstanten Element als Argument.

Tabelle 7.54 Weitere Methoden

Ein Beispiel, das einige dieser Methoden in der Praxis zeigen soll:

```
// valarray2.cpp
#include <iostream>
#include <valarray>
using namespace std;

int verdoppeln( const int val );

int main( int argc, char **argv ) {
   // Ein valarray mit 5 Elementen vom Typ int
   valarray<int> va1(5);
   // Array initialisieren
   for( unsigned int i=0; i < va1.size(); i++ ) {
      va1[i] = i*i;
   }
   // Alle Elemente ausgeben
   cout << "va1      : ";
   for( unsigned int i=0; i < va1.size(); i++ ) {
      cout << va1[i] << " ";
   }
   cout << endl;
   // Summe aller Elemente
   cout << "Summe va1: " << va1.sum() << endl;
   // Kleinstes Element
   cout << "Kleinste : " << va1.min() << endl;
   // Größtes Element
   cout << "Größte   : " << va1.max() << endl;
   // Ring-Shift nach vorne
   valarray<int> va2( va1.cshift(2) );
   // Alle Element ausgeben
   cout << "va2      : ";
   for( unsigned int i=0; i < va2.size(); i++ ) {
      cout << va2[i] << " ";
   }
   cout << endl;
   // Ein Ring-Shift zurück
   va2 = va1.cshift(-5);
   // Alle Elemente ausgeben
   cout << "va2      : ";
   for( unsigned int i=0; i < va2.size(); i++ ) {
      cout << va2[i] << " ";
   }
   cout << endl;

   // Alle Elemente verdoppeln
```

```
   va2 = va1.apply( verdoppeln );
   // Alle Elemente ausgeben
   cout << "va2      : ";
   for( unsigned int i=0; i < va2.size(); i++ ) {
      cout << va2[i] << " ";
   }
   cout << endl;
   return 0;
}

int verdoppeln ( const int val ) {
   return val*2;
}
```

Das Programm bei der Ausführung:

```
va1      : 0 1 4 9 16
Summe va1: 30
Kleinste : 0
Größte   : 16
va2      : 4 9 16 0 1
va2      : 0 1 4 9 16
va2      : 0 2 8 18 32
```

Operatoren

Die Klasse `valarray` definiert auch alle gewöhnlichen Operatoren. Dies schließt die Addition, Subtraktion, Multiplikation, Division, Modulo, Negation, Bit-Operatoren, Vergleichsoperatoren, unäre Operatoren und die logischen Operatoren mit ein. Dies gilt auch für den Zuweisungsoperator. Dabei sind zwei Möglichkeiten implementiert – zunächst die Verwendung für Skalare, die Sie bereits von den gewöhnlichen Arrays her kennen und, das ist neu, die Verwendung für Vektoren. Beispielsweise ist es neben einfachen skalaren Berechnungen wie

```
va1[0] = va2[0] + va4[0];
```

auch möglich, mit folgendem Ausdruck den ganzen Vektor zu berechnen:

```
va1 = va2 + va4;
```

Dieser Ausdruck ist gleichwertig zu

```
va1[0] = va2[0] + va4[0];
va1[1] = va2[1] + va4[1];
va1[2] = va2[2] + va4[2];
...
va1[n] = va2[n] + va4[n];
```

Beachten Sie allerdings, dass das Ergebnis nicht mehr definiert ist, sobald die Anzahl der Elemente zwischen den einzelnen `valarray` variiert. Diesen eben gezeigten Ausdruck können Sie selbstverständlich auch mit jedem beliebigen anderen Operator durchführen.

Natürlich lassen sich hierbei auch binäre Operationen durchführen, bei denen einer der Operanden ein einfacher numerischer Wert ist – das Mischen eines Vektors und eines Skalars und umgekehrt. Beispielsweise ist folgender Ausdruck

```
va1 = 5 * va4;
// oder
// val1 = va4 * 5;
```

gleichwertig zu

```
va1[0] = 5 * va4[0];
va1[1] = 5 * va4[1];
va1[2] = 5 * va4[2];
...
va1[n] = 5 * va4[n];
```

Neben der gewöhnlichen Verwendung der global definierten Operatoren, mit denen Sie zwei Vektoren bzw. einen Vektor und ein Skalar miteinander verknüpfen können, steht Ihnen auch die zusammengesetzte Zuweisung zur Verfügung (auch wieder in beiden Versionen) wie beispielsweise:

```
va1 = va1 + va2;
// oder als zusammengesetzte Zuweisung ...
va1 += va2;
```

Bei der Verwendung von Vergleichsoperatoren wird allerdings mit einem Vergleich wie

```
va1 == va2
```

kein boolescher Wert zurückgegeben, sondern ein neues `valarray`, in dem alle Elemente vom Typ `bool` sind. Daher benötigen Sie einen Ausdruck wie

```
valarray<bool> va_bool(5);
...
va_bool = (va2 == va3);
```

Dieser Ausdruck ist gleichwertig zu

```
va_bool[0] = ( va2[0] == va3[0] );
va_bool[1] = ( va2[1] == va3[1] );
va_bool[2] = ( va2[2] == va3[2] );
...
va_bool[n] = ( va2[n] == va3[n] );
```

Hierzu ein Listing, das die Verwendung von Operatoren mit der Klasse `valarray` demonstrieren soll:

```
// valarray3.cpp
#include <iostream>
#include <valarray>
using namespace std;

int main( int argc, char **argv ) {
   // Ein valarray mit 5 Elementen vom Typ int
   valarray<int> va1(5);
   // Ein valarray mit 5 Elementen vom Typ int,
   // die mit dem Wert 1 initialisiert werden
   valarray<int> va2(1, 5);
   // Kopierkonstruktor
   valarray<int> va3( va2 );
   // Ein C-Array
   int Carr[] = { 1, 2, 4, 8, 16 };
   // Ein valarray aus dem C-Array machen
   valarray<int> va4(Carr, sizeof(Carr)/sizeof(int));
   valarray<bool> vab(5);

   va1 = va2 + va4;
   cout << endl << "va1: ";
   for( unsigned int i=0; i < va1.size(); i++ ) {
      cout << va1[i] << " ";
   }
   va1 = 5 * va4;
   cout << endl << "va1: ";
   for( unsigned int i=0; i < va1.size(); i++ ) {
      cout << va1[i] << " ";
   }
   vab = (va2 == va3);
   cout << endl << "va2 == va3: ";
   bool checkbool = true;
   for( unsigned int i=0; i < va2.size(); i++ ) {
      if( vab[i] != true )
         checkbool = false;
   }
   if( checkbool == true ) {
      cout << "sind beide gleich" << endl;
   }
   else {
      cout << "sind nicht gleich" << endl;
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
va1: 2 3 5 9 17
va1: 5 10 20 40 80
va2 == va3: sind beide gleich
```

Mathematische Funktionen

Wem nützt ein auf Zahlen spezialisierter Vektor, wenn es keine mathematischen Funktionen dafür gibt. Daran haben auch die Entwickler gedacht und die wichtigsten Funktionen für Berechnungen aller Art spendiert (siehe Tabelle 7.55). Alle Methoden verwenden ein `valarray` als Argument und geben auch wieder ein `valarray` zurück.

Funktion	Beschreibung
`template<class T> valarray<T> abs (const valarray<T>& v);`	Gibt für jedes Element von `v` den Absolutbetrag zurück und schreibt diesen in das entsprechende Element des Ergebnis-Vektors.
`template<class T> valarray<T> acos (const valarray<T>& v);`	Berechnet für jedes Element von `v` den entsprechenden Funktionswert und schreibt diesen in das entsprechende Element des Ergebnisvektors.
`template<class T> valarray<T> asin (const valarray<T>& v);`	
`template<class T> valarray<T> atan (const valarray<T>& v);`	
`template<class T> valarray<T> cos (const valarray<T>& v);`	
`template<class T> valarray<T> cosh (const valarray<T>& v);`	
`template<class T> valarray<T> exp (const valarray<T>& v);`	
`template<class T> valarray<T> log (const valarray<T>& v);`	
`template<class T> valarray<T> log10(const valarray<T>& v);`	
`template<class T> valarray<T> sin (const valarray<T>& v);`	
`template<class T> valarray<T> sinh (const valarray<T>& v);`	
`template<class T> valarray<T> sqrt (const valarray<T>& v);`	
`template<class T> valarray<T> tan (const valarray<T>& v);`	
`template<class T> valarray<T> tanh (const valarray<T>& v);`	

Tabelle 7.55 Mathematische Funktionen

Neben diesen Funktionen gibt es noch die zwei überladenen Funktionen `atan2()` und `pow()`:

Funktion	Beschreibung
`template<class T> valarray<T> atan2( const valarray<T>& v, const valarray<T>& w);`	Berechnet für jedes Element `v[i]` von `v` und `w[i]` von `w` den Arcus Tangens von `v[i]/w[i]` und schreibt den Funktionswert in das entsprechende Element des Ergebnisvektors.
`template<class T> valarray<T> atan2( const valarray<T>& v, const T& t);`	Berechnet für jedes Element `v[i]` von `v` den Arcus Tangens von `v[i]/t` und schreibt den Funktionswert in das entsprechende Element des Ergebnisvektors.
`template<class T> valarray<T> atan2( const T& t, const valarray<T>& v);`	Berechnet für den Wert `t` und jedes Element `v[i]` von `v` den Arcus Tangens von `t/v[i]` und schreibt den Funktionswert in das entsprechende Element des Ergebnisvektors.
`template<class T> valarray<T> pow( const valarray<T>& v, const valarray<T>& w);`	Potenziert das Element von `v` mit dem entsprechenden Element von `w` und schreibt das Ergebnis in das entsprechende Element des Ergebnisvektors.
`template<class T> valarray<T> pow(const valarray<T>& v, const T& t);`	Potenziert das Element von `v` mit `t` und schreibt das Ergebnis in das entsprechende Element des Ergebnisvektors.
`template<class T> valarray<T> pow( const T& t, const valarray<T>& v);`	Potenziert `t` mit jedem Element von `v` und schreibt das Ergebnis in das entsprechende Element des Ergebnisvektors.

Tabelle 7.56 Weitere überladene mathematische Funktionen

Arbeiten mit Teilvektoren

Der Zugriffsoperator des Templates `valarray<T>` wurde so überladen, dass auch Teilvektoren angesprochen werden können. Ein solcher Teilvektor wird mit Hilfe einer Teilmenge der Indexmenge des Vektors ausgewählt (es wurde bereits erwähnt, dass die Verwendung von `valarray` ein wenig umständlich ist). Eine solche Auswahl der Teilmenge der Indexwerte kann auf vier verschiedene Arten erfolgen:

- Slices (`slice`)
- General Slices (`gslice`)
- Masked Subsets (`mask_array`)
- Indirect Subsets (`indirect_array`)

Die Klasse »slice«

`slice` (dt.: Teil, Schnitte) ermöglichen die selektive Indizierung von Vektorelementen, wobei ein Teil der Indexmenge eines Vektors ausgewählt wird. Jedes Objekt vom Typ `slice` wird durch einen Startindex, eine Schrittweite und die Anzahl der Elemente bestimmt. Damit ist es möglich, jede Zeile oder Spalte eines `valarray`, das als Matrix interpretiert wird, zu beschreiben oder andere Teilmengen eines `valarray` anzusprechen.

Hinweis **[«]**

In der linearen Algebra ist eine Matrix (Plural: Matrizen) eine Anordnung von Zahlen (oder anderen Objekten) in Tabellenform. Man spricht von den Spalten und Zeilen der Matrix, sie bilden Zeilen- bzw. Spaltenvektoren. Die Elemente, die in der Matrix angeordnet sind, nennt man *Einträge* oder *Komponenten* der Matrix.

Beispielsweise definieren Sie mit

```
// slice s ( Startindex, N_Elemente, Schrittweite );
slice s( 3, 4, 5 );
```

ein Objekt `s` vom Typ `slice`, das mit dem Index 3 beginnt und 4 Indizes bei einer Schrittweite von 5 auswählt. Das wären dann diese Indizes:

```
// Startindex:3, Elemente:4, Schrittweite:5
3, 8, 13, 18
```

In Bezug zum `valarray` wird dabei der Zugriffsoperator `[]` verwendet, der auch für die Klasse `slice` überladen ist. Damit werden praktisch folgende Operationen möglich:

```
va[slice(0,4,3)] =
   valarray<int>(va[std::slice (1,4,3)]) *
   valarray<int>(va[std::slice (2,4,3)]);
```

Betrachtet man dieses Beispiel an einer 4×3-Matrix, so ist diese Berechnung gleichwertig zu:

```
va[0] = va[1] * va[2];
va[3] = va[4] * va[5];
va[6] = va[7] * va[8];
va[9] = va[10] * va[11];
```

Abbildung 7.3 soll zeigen, was man sich darunter vorstellen kann.

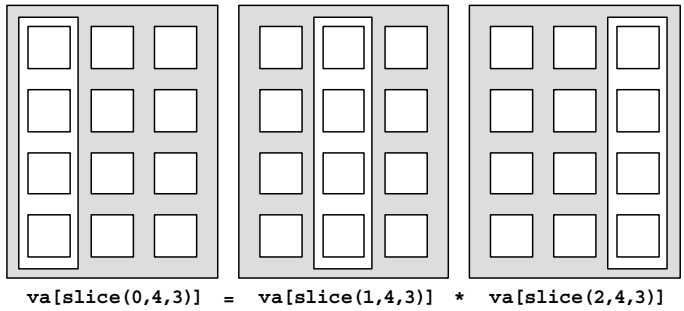

Abbildung 7.3 Eine einfache Matrix-Berechnung

Versuchen Sie einmal zu erraten, in welche »Scheibchen« wir die folgende Berechnung zerschnitten haben:

```
// 4×3-Matrix
va2[slice(0,3,1)] =
   valarray<int>(va2[slice (3,3,1)]) +
   valarray<int>(va2[slice (6,3,1)]) +
   valarray<int>(va2[slice (9,3,1)]);
```

Hier haben Sie eine zeilenweise Addition, wie Abbildung 7.4 zeigen soll:

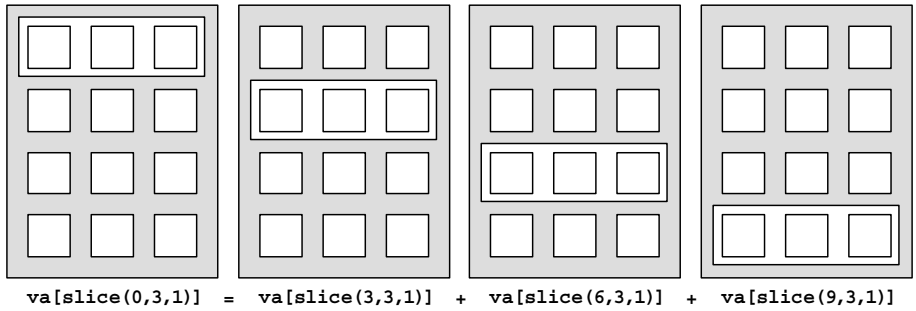

Abbildung 7.4 Eine einfache zeilenweise Addition

Hierzu ein Beispiel, das die Verwendung von `slice` in der Praxis demonstriert:

```
// valarray4.cpp
#include <iostream>
#include <valarray>
using namespace std;

template<class T>
void print (const valarray<T>& va, int num);
```

```
int main( int argc, char **argv ) {
   int zeile = 4;
   int spalte= 3;
   // valarray mit 12 Elementen (4 Zeilen, 3 Spalten)
   valarray<int> va(zeile*spalte);
   // Initialisieren ...
   for (int i=0; i<zeile*spalte; i++) {
       va[i] = i;
   }
   // Kopieren ...
   valarray<int> va2( va );
   // Ausgeben ...
   print( va, spalte );
   // Multiplikation
   va[slice(0,4,3)] =
      valarray<int>(va[slice (1,4,3)]) *
      valarray<int>(va[slice (2,4,3)]);

   // Ausgeben ...
   print( va, spalte );

   va2[slice(0,3,1)] =
      valarray<int>(va2[slice (3,3,1)]) +
      valarray<int>(va2[slice (6,3,1)]) +
      valarray<int>(va2[slice (9,3,1)]);
   // Ausgeben ...
   print( va2, spalte );
   return 0;
}

// Gibt das valarray Zeile für Zeile aus
template<class T>
void print (const valarray<T>& va, int num) {
   for (unsigned int i=0; i < va.size()/num; ++i) {
      for (int j=0; j < num; ++j) {
         cout << va[i*num+j] << ' ';
      }
      cout << endl;
   }
   cout << endl;
}
```

Das Programm bei der Ausführung:

```
0 1 2
3 4 5
6 7 8
9 10 11

2 1 2
20 4 5
56 7 8
110 10 11

18 21 24
3 4 5
6 7 8
9 10 11
```

Die Klasse »gslice«

Im Beispiel mit `slice` konnten Sie sehen, wie man Zeilen und Spalten von einem zweidimensionalen Feld beschreiben kann. Benötigen Sie hingegen einen Teilvektor, der keine einzelne Zeile oder Spalte ist, können Sie die Klasse `gslice` verwenden. Mit dieser können Sie zum Beispiel aus einer 4×3-Matrix eine 2×2-Matrix bilden.

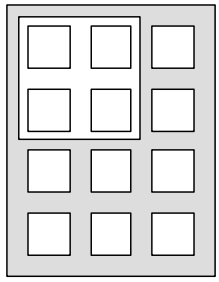

Abbildung 7.5 Teilvektor mit »gslice« aus einer Matrix extrahieren

Ein `gslice` enthält fast dieselben Informationen wie ein `slice`, nur werden nicht ein Abstand und eine Anzahl wie bei `slice`, sondern `n` Abstände und `n` Anzahlen gespeichert. Also statt wie bei `slice` mit

```
slice s ( Startindex, N_Elemente, Schrittweite );
```

werden für die Parameter zwei und drei je ein Längen- und ein Abstandspaar verwendet:

```
gslice gs( Startindex, LängenPaar, Abstandspaar );
```

Die Syntax zu `gslice` zeigt, dass die beiden Paare wiederum vom Typ `valarray` sind:

```
gslice gs ( size_t s,
            const valarray<size_t>& l,
            const valarray<size_t>& d );
```

Als Beispiel sei eine 3×9-Matrix gegeben (siehe Abbildung 7.6):

```
int zeile = 3;
int spalte= 9;
// valarray mit 27 Elementen (3 Zeilen, 9 Spalten)
valarray<int> vi(zeile*spalte);
// Initialisieren ...
for (int i=0; i<zeile*spalte; i++) {
    vi[i] = i;
}
```

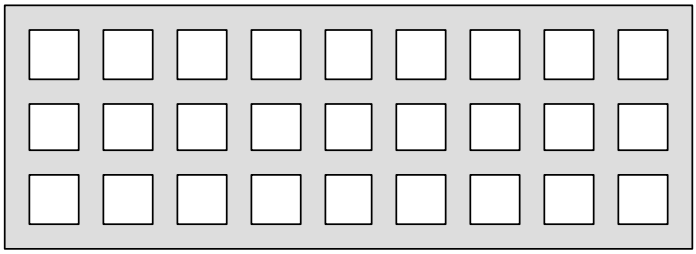

Abbildung 7.6 3×9-Matrix

Wollen Sie hier alle Elemente auswählen, müssen Sie jeweils das Längen- und Abstandspaar beschreiben. Dies können Sie folgendermaßen machen:

```
// length_buf[0],stride_buf[0] beschreiben die Zeile
// length_buf[1],stride_buf[1] beschreiben die Spalte
size_t length_buf[] = { 3,9 };
size_t stride_buf[] = { 9,1 };
```

Mit diesen Zeilen spezifizieren Sie eine Länge von 3 (`length_buf[0]`) und einen Abstand von 9 (`stride_buf[0]`) für eine Zeile. Dementsprechend spezifizieren Sie die Länge von 9 (`length_buf[1]`) und einen Abstand von 1 (`stride_buf[1]`) für eine Spalte. Damit haben Sie eine 3×9-Matrix beschrieben. Allerdings kann `gslice` nichts mit diesen Werten anfangen, da ein Parameter vom Typ `valarray<size_t>` verlangt wird. Nichts einfacher als das:

```
// wird für gslice als zweiter Parameter benötigt
valarray<size_t> length_val(length_buf, 2);
// wird für gslice als dritter Parameter benötigt
```

```
valarray<size_t> stride_val(stride_buf, 2);
// Startindex:0
size_t start = 0;
```

Jetzt können Sie mit folgendem Code die komplette Matrix 3×9 mit dem Kopierkonstruktor kopieren (unsinnig, aber möglich):

```
valarray<int> vi2(vi[gslice(start,length_val,stride_val)]);
```

Sicherlich werden Sie nicht auf so umständliche Weise eine komplette Matrix kopieren. Daher schneiden wir jetzt einfach einen Teil aus der Mitte der Matrix aus:

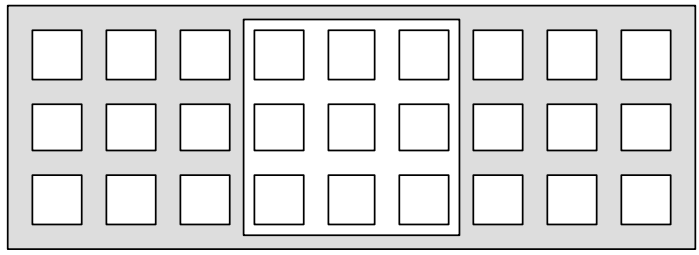

Abbildung 7.7 Teil-Array extrahieren

Als Startindex haben wir zunächst die 3. Aber wie beschreibt man den Rest? Zunächst müssen Sie die Zeile beschreiben. Hier haben Sie also wie gehabt eine Länge von 3 (`length_buf[0]`) und einen Abstand von 9 (`stride_buf[0]`). Für die Länge benötigen wir hingegen nur eine Länge von 3 (`length_buf[1]`) und einen Abstand von 1 (`stride_buf[1]`), und schon haben Sie die Angaben für die Spalte. In Zahlen sieht dies folgendermaßen aus:

```
gslice(3,{3,3},{9,1})
```

Programmtechnisch lässt sich dies so umsetzen:

```
size_t length_buf2[] = {3, 3};
size_t stride_buf2[] = {9, 1};
valarray<size_t> length_val2(length_buf2, 2);
valarray<size_t> stride_val2(stride_buf2, 2);
size_t start = 3;
valarray <int> vi4(
   vi[std::gslice(start,length_val2,stride_val2)] );
```

Natürlich ist es auch möglich, eine Matrix in mehrere Scheibchen senkrecht, waagerecht oder auch diagonal zu »zerschneiden«. Beispielsweise lässt sich die 3×9-Matrix mit

```
gslice(0,{3,3},{3,10})
```

in diagonale Scheibchen (siehe Abbildung 7.8) zerteilen.

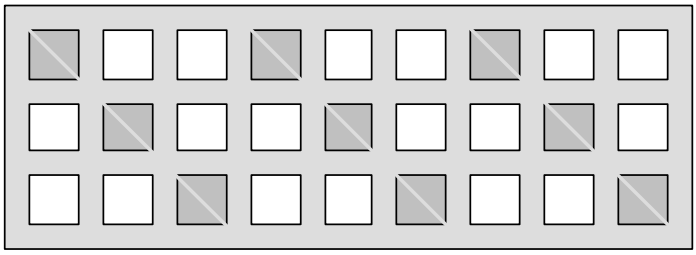

Abbildung 7.8 Diagonale Teil-Arrays extrahieren

Hierzu noch ein komplettes Listing, das das Extrahieren von Teil-Arrays mit `gslice` in der Praxis demonstriert:

```
// valarray5.cpp
#include <valarray>
#include <iostream>
using namespace std;

typedef valarray<int>    valarray_t;
typedef valarray<size_t> selector_t;
template<class T>
void print (const valarray<T>& va, int num);

int main(void) {
   int zeile = 3;
   int spalte= 9;
   // valarray mit 27 Elementen (3 Zeilen, 9 Spalten)
   valarray_t vi(zeile*spalte);
   // Initialisieren ...
   for (int i=0; i<zeile*spalte; i++) {
       vi[i] = i;
   }

  // Ausgeben
  print( vi, 9 );

  // length_buf[0],stride_buf[0] beschreiben die Zeile
  // length_buf[1],stride_buf[1] beschreiben die Spalte
  size_t length_buf[] = { 3,9 };
  size_t stride_buf[] = { 9,1 };
  // wird für gslice als zweiter Parameter benötigt
  selector_t length_val(length_buf, 2);
```

```
// wird für gslice als dritter Parameter benötigt
selector_t stride_val(stride_buf, 2);
// Startindex:0
size_t start = 0;

// komplette Matrix ausgeben ...
// gslice(0,{3,9},{9,1})
valarray_t vi2(vi[ gslice(start,length_val,stride_val) ]);
print( vi2, spalte );

// Nur die zweite Zeile selektieren
size_t length_buf2[] = {3, 3};
size_t stride_buf2[] = {3, 1};
selector_t             length_val2(length_buf2, 2);
selector_t             stride_val2(stride_buf2, 2);
start = 9;
// gslice(9,{3,3},{3,1})
valarray_t vi3(
   vi[std::gslice(start,length_val2,stride_val2)]);
print( vi3, spalte );

// Wir schneiden etwas aus der Mitte aus ...
stride_val2[0] = 9;
stride_val2[1] = 1;
start = 3;
// gslice(3,{3,3},{9,1})
valarray_t vi4(
   vi[std::gslice(start,length_val2,stride_val2)] );
print( vi4, 3 );

// Mehrere Teile senkrecht "ausschneiden"
stride_val2[0] = 3;
stride_val2[1] = 9;
start = 1;
// gslice(1,{3,3},{3,9})
valarray_t vi5(
   vi[std::gslice(start,length_val2,stride_val2)] );
print( vi5, 3 );

// Jetzt in diagonale Scheibchen "zerschneiden"
stride_val2[0] = 3;
stride_val2[1] = 10;
start = 0;
// gslice(0,{3,3},{3,10})
valarray_t vi6(
```

```
     vi[std::gslice(start,length_val2,stride_val2)]  );
  print( vi6, 3 );
  return 0;
}

// Gibt das valarray Zeile für Zeile aus
template<class T>
void print (const valarray<T>& va, int num) {
   for (unsigned int i=0; i < va.size()/num; ++i) {
      for (int j=0; j < num; ++j) {
         cout.width(2);
         cout << va[i*num+j] << ' ';
      }
      cout << endl;
   }
   cout << endl;
}
```

Das Programm bei der Ausführung:

```
 0  1  2  3  4  5  6  7  8
 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26

 0  1  2  3  4  5  6  7  8
 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26

 9 10 11 12 13 14 15 16 17

 3  4  5
12 13 14
21 22 23

 1 10 19
 4 13 22
 7 16 25

 0 10 20
 3 13 23
 6 16 26
```

Die Klasse »mask_array«

Mit der Klasse `mask_array` haben Sie eine weitere Möglichkeit, eine Teilmenge eines `valarray` zu beschreiben, um das Resultat wie ein `valarray` erscheinen zu lassen. Ein solches `mask_array` ist im Grunde nur ein `valarray<bool>`. Verwen-

den Sie einen solchen Index eines `valarray`, bedeutet `true`, dass ein entsprechendes Element Teil des Ergebnisses sein soll. Damit können Sie praktisch aus einer Matrix Teilmengen extrahieren, für die es einfach kein sinnvolles Muster mehr gibt.

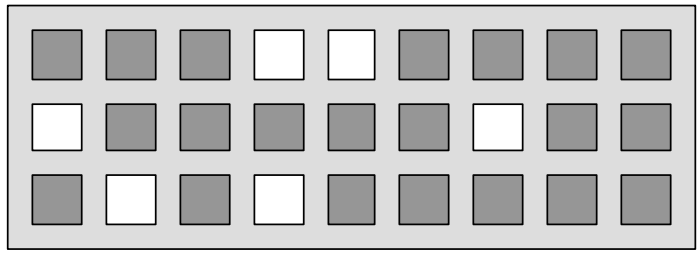

Abbildung 7.9 Dunkle Felder sollen extrahiert werden.

In Abbildung 7.9 kann man erkennen, dass es hier kein logisches Muster gibt. Hierzu kann man am besten die 3×9-Matrix mit einem `bool`-Array maskieren. Das sieht in der Praxis wie folgt aus:

```
bool vbool[] = {
   true, true, true, false, false, true, true, true, true,
   false, true, true, true, true, true, false, true, true,
   true, false, true, false, true, true, true, true, true };

valarray<bool> vb1( vbool, 27 );
```

Mit dieser Maske können Sie nun aus der 3×9-Matrix die Elemente mit dem Indexoperator extrahieren:

```
valarray<int> vi2( vi[vb1] );
```

Natürlich muss die Anzahl der Elemente in einem `valarray` mit der Anzahl der Elemente übereinstimmen (genauer: gleich sein), auf dem die Maske verwendet wird.

Hierzu wieder ein komplettes Listing, das bestimmte Teilmengen mit Hilfe einer Maske aus einer 3×9-Matrix extrahiert:

```
// valarray6.cpp
#include <valarray>
#include <iostream>
using namespace std;
typedef valarray<int>    valarray_t;
typedef valarray<bool>   valarray_b;
typedef valarray<size_t> selector_t;
template<class T>
void print (const valarray<T>& va, int num);
```

```
int main(void) {
   int zeile = 3;
   int spalte= 9;
   // valarray mit 27 Elementen (3 Zeilen, 9 Spalten)
   valarray_t vi(zeile*spalte);
   // auch eine Maske mit 3*9 Elementen erzeugen
   valarray_b vb2(zeile*spalte);

   bool vbool[] = {
     true, true, false, false, true, true, true, true, true,
     false, true, true, true, true, true, false, true, true,
     true, false, true, false, true, true, true, true, true
   };
   // noch eine 3*9 Matrix
   valarray_b vb1( vbool, 27 );

   for( int i = 0; i<zeile*spalte; i++ ) {
      vi[i] = i;
   }
   // Immer drei Elemente maskieren und den Abstand
   // um 1 erhöhen
   for( int i = 0, j = 0; i<zeile*spalte; i+=j ) {
      vb2[i++] = true;
      vb2[i++] = true;
      vb2[i++] = true;
      j++;
   }
   // Ausgeben
   print( vi, 9 );
   print( vb1, 9 );
   print( vb2, 9 );

   // Neues valarray mit Hilfe einer Maske erzeugen
   valarray_t vi2( vi[vb1] );
   print( vi2, 7 );
   valarray_t vi3( vi[vb2] );
   print( vi3, 5 );
   return 0;
}
// Gibt das valarray Zeile für Zeile aus
template<class T>
void print (const valarray<T>& va, int num) {
   for (unsigned int i=0; i < va.size()/num; ++i) {
```

```
      for (int j=0; j < num; ++j) {
         cout.width(2);
         cout << va[i*num+j] << ' ';
      }
      cout << endl;
   }
   cout << endl;
}
```

Das Programm bei der Ausführung:

```
 0  1  2  3  4  5  6  7  8
 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26

 1  1  0  0  1  1  1  1  1
 0  1  1  1  1  1  0  1  1
 1  0  1  0  1  1  1  1  1

 1  1  1  0  1  1  1  0  0
 1  1  1  0  0  0  1  1  1
 0  0  0  0  1  1  1  0  0

 0  1  4  5  6  7  8
10 11 12 13 14 16 17
18 20 22 23 24 25 26

 0  1  2  4  5
 6  9 10 11 15
16 17 22 23 24
```

[»] Hinweis

Ein `mask_array` kann nicht direkt erzeugt werden, sondern wird als Ergebnis des Einsatzes eines `valarray<bool>` als Index des `valarray` verwendet.

Die Klasse »indirect_array«

Mit der Klasse `indirect_array` können Sie ähnlich wie mit dem `mask_array` eine beliebige Teilmenge von `valarray` extrahieren. Nur geschieht hier die Adressierung der einzelnen Elemente indirekt über den Index des Arrays. Ein `indirect_array` kann, wie schon ein `mask_array`, nicht direkt erzeugt werden und wird über die Verwendung von `valarray<size_t>` als Index eines `valarray` realisiert. Das Prinzip über die indirekte Adressierung ist recht einfach, weshalb ich mir hierzu eine umfassendere Erläuterung sparen kann und gleich ein Beispiel dazu liefere:

```
// valarray7.cpp
#include <valarray>
#include <iostream>
using namespace std;

typedef valarray<int>    valarray_t;
typedef valarray<size_t>   valarray_indirect;
typedef valarray<size_t> selector_t;
template<class T>
void print (const valarray<T>& va, int num);

int main(void) {
   int zeile = 3;
   int spalte= 9;
   // valarray mit 27 Elementen (3 Zeilen, 9 Spalten)
   valarray_t vi(zeile*spalte);
   valarray_indirect vindirect1(zeile*spalte);
   // Elemente 2, 1, 4, 3, 6, 5, 8, 7, 9 extrahieren
   size_t vindbuf[] = { 2, 1, 4, 3, 6, 5, 8, 7, 9 };
   valarray_indirect vindirect2(vindbuf, 9);

   for( int i = 0; i<zeile*spalte; i++ ) {
      vi[i] = i;
   }
   // Die Reihenfolge umdrehen
   for( int i = zeile*spalte-1, j=0; i>=0; i--, j++) {
      vindirect1[j] = i;
   }
   valarray_t vi2( vi[vindirect1] );
   valarray_t vi3( vi[vindirect2] );

   print( vi, 9 );
   print( vi2, 9 );
   print( vi3, 3 );
   return 0;
}

// Gibt das valarray Zeile für Zeile aus
template<class T>
void print (const valarray<T>& va, int num) {
   for (unsigned int i=0; i < va.size()/num; ++i) {
      for (int j=0; j < num; ++j) {
         cout.width(2);
         cout << va[i*num+j] << ' ';
      }
      cout << endl;
```

```
    }
    cout << endl;
}
```

Das Programm bei der Ausführung:

```
 0  1  2  3  4  5  6  7  8
 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26

26 25 24 23 22 21 20 19 18
17 16 15 14 13 12 11 10  9
 8  7  6  5  4  3  2  1  0

 2  1  4
 3  6  5
 8  7  9
```

7.3.3 Globale numerische Funktionen (»cmath« und »cstdlib«)

Aus der Programmiersprache C stehen Ihnen ebenfalls weitere globale numerische Funktionen zur Verfügung. Die meisten Funktionen sind in der Header-Datei <*cmath*> (in C als <*math.h*> bekannt) definiert. Aber auch in der Header-Datei <*cstdlib*> (alias <*stdlib.h*> in C) sind einige mathematische Routinen vorhanden. Damit dieses Buch komplett ist, sollen die darin enthaltenen mathematischen Funktionen kurz beschrieben werden.

<cmath>

[»] **Hinweis für Linux-User**

Damit ein Programm die <cmath>-Bibliothek verwenden kann, muss diese häufig erst mit dem Compiler-Flag `-lm` hinzugelinkt werden. Beispiel: `g++ –o programm programm.c –lm`

Hier ein Überblick über die Funktionen in der Header-Datei <*cmath*> und deren Bedeutungen:

Funktion	Beschreibung
`double` **`acos`**`(double zahl)`	Arcuscosinus
`double` **`asin`**`(double zahl)`	Arcussinus
`double` **`atan`**`(double zahl)`	Arcustangens
`double` **`atan2`**`(double zahl1, double zahl2)`	Arcustangens von `zahl1` und `zahl2`

Tabelle 7.57 Mathematische Funktionen

Funktion	Beschreibung
`double cos(double zahl)`	Cosinus
`double sin(double zahl)`	Sinus
`double tan(double zahl)`	Tangens
`double cosh(double zahl)`	Cosinus hyperbolicus
`double sinh(double zahl)`	Sinus hyperbolicus
`double tanh(double zahl)`	Tangens hyperbolicus
`double exp(double zahl)`	Exponentialfunktion berechnen
`double log(double zahl)`	Logarithmus von `zahl` zur Basis e = 2.71828...
`double log10(double zahl)`	Logarithmus zur Basis 10
`double sqrt(double zahl)`	Quadratwurzel
`double ceil(double zahl)`	Gleitpunktzahl aufrunden
`double fabs(double zahl)`	Absolutwert
`double floor(double zahl)`	Gleitpunktzahl abrunden
`double frexp(double zahl, int zahl2)`	Zerlegt `zahl` in eine Mantisse und einen ganzzahligen Exponenten.
`double modf(double zahl1, double *zahl2)`	Zerlegt den Wert von `zahl1` in einen gebrochenen und einen ganzzahligen Wert. Der ganzzahlige Wert (Vorkommateil) befindet sich dann in der Adresse von `zahl2`.
`double pow(double zahl1, double zahl2)`	Potenz `zahl1^zahl2`
`int fmod(double zahl1, double zahl2)`	`float modulo` errechnet den Rest von `zahl1/zahl2`.

Tabelle 7.57 Mathematische Funktionen (Forts.)

Es fällt außerdem auf, dass all diese Funktionen mit dem Typ `double` deklariert sind. Abhilfe schafft da allerdings erst der ANSI-C99-Standard.

<cmath> – C99-Standard

Hinweis [«]

Es kann sein, dass dieser erweiterte ANSI-C99-Standard bei vielen Compilern noch gar nicht implementiert ist und deshalb auch nicht funktioniert. Wir beziehen uns auf C, aber wie erwähnt: C ist eine Untermenge von C++.

Die Header-Datei *<cmath>* wurde mit der Einführung des C99-Standards sehr erweitert. In ihr befinden sich jetzt noch mehr Funktionen und Makros. Zu allen Funktionen wurden Versionen herausgebracht, die jetzt auch auf die Datentypen `float` und `long double` anwendbar sind. Bisher waren die Funktionen dieser

Header-Datei ja nur mit `double` angegeben. Um die für den Datentyp passende Funktion zu verwenden, müssen Sie nur ein entsprechendes Suffix notieren. `f` steht für `float`, `l` für `long double` und keine Angabe steht – wie gehabt – für `double`-Gleitpunktzahlen. Als Beispiel hier die Funktion `sqrt()`:

```
float sqrtf(float zahl);
double sqrt(double zahl);
long double sqrtl(long double zahl);
```

In der folgenden Tabelle finden Sie einige Funktionen, die in der Header-Datei *<cmath>* neu hinzugekommen sind. Zu all diesen Funktionen gibt es auch schon verschiedene Versionen. Es hängt davon ab, welches Suffix Sie verwenden.

Funktion	Bedeutung

`double round (double);`
`double trunc (double);`
`double rint (double x)`
	Funktionen zum Runden von Zahlen

`double fmax (double, double);` `double fmin (double, double);`	Maximum, Minimum
`double log2 (double _x);` `double logb (double);`	Logarithmus
`double copysign (double,` `double);`	Vorzeichen kopieren

`double scalb (double, long);`
`extern double fma (double,`
`double, double);`
	laufzeitoptimierte Berechnungen

`double hypot (double, double);`	Wurzel

Tabelle 7.58 Neue mathematische Funktionen

Sehr interessant dürften die Makros für den Vergleich von Gleitpunktzahlen sein, die ebenfalls hinzugekommen sind:

Makro	Bedeutung
`isgreater(x, y)`	`x` größer als `y`
`isgreaterequal(x, y)`	`x` größer als oder gleich `y`
`isless(x, y)`	`x` kleiner als `y`
`islessequal(x, y)`	`x` kleiner als oder gleich `y`
`islessgreater(x, y)`	`x` kleiner als `y` ODER `x` größer als `y`
`isunordered(x, y)`	Sind `x` und `y` nicht miteinander vergleichbar, gibt dieses Makro `1` zurück, ansonsten `0`.

Tabelle 7.59 Makros zum Vergleichen von Gleitpunktzahlen

Ein weiteres interessantes Feature sind Makros zur Bestimmung der Kategorie von Gleitpunktzahlen. In ANSI C werden die Gleitpunktzahlen in folgende fünf Kategorien unterteilt (Konstanten aus der Header-*Datei <cmath>*):

Konstante	Kategorie
`FP_NAN`	`NAN` steht für *Not a Number* und bedeutet, dass es sich bei dem Wert um keine gültige Gleitpunktdarstellung handelt.
`FP_NORMAL`	eine Gleitpunktzahl in normaler Darstellung
`FP_INFINITE`	Die Gleitpunktzahl wird als unendlicher Wert dargestellt.
`FP_ZERO`	eine Gleitpunktzahl mit dem Wert `0`
`FP_SUBNORMAL`	eine Gleitpunktzahl, mit der besonders kleine Zahlen dargestellt werden können

Tabelle 7.60 Bestimmung der Gleitpunktzahl-Kategorie

Abfragen, in welche Kategorie eine bestimmte Gleitpunktzahl fällt, können Sie mit den folgenden Makros vornehmen:

Makro	Bedeutung
`isnan(x)`	Ist die Zahl gleich `FP_NAN`, wird `1` zurückgegeben, ansonsten `0`.
`isnormal(x)`	Ist die Zahl gleich `FP_NORMAL`, wird `1` zurückgegeben, ansonsten `0`.
`isfinite(x)`	Ist die Zahl eine endliche, wird `1` zurückgegeben, ansonsten `0`.
`isinf(x)`	Ist die Zahl gleich `FP_INFINITE`, wird `1` zurückgegeben, ansonsten `0`.

Tabelle 7.61 Makros zur Bestimmung der Gleitpunktzahl-Kategorie

Intern werden alle diese Makros jedoch zum Teil mit Hilfe des Makros `fpclassify()` ausgewertet:

```
fpclassify(x) == FP_INFINITE //isinf(x)
fpclassify(x) == FP_NORMAL   //isnormal(x)
```

<cstdlib> – mathematische Funktionen

Funktion	Beschreibung
`int abs(int x);` `long labs(long x);`	Sie erhalten den Absolutwert zum ganzzahligen Argument `x`.

`div_t div(`
`  int zaehler,`
`  int nenner);`
`ldiv_t ldiv(`
`  long int zaehler,`
`  long int nenner );`
	Berechnet den Quotienten und den Rest einer Division.

Tabelle 7.62 Mathematische Funktionen der Header-Datei <cstdlib>

Funktion	Beschreibung
`int rand();`	Liefert eine Pseudozufallszahl zwischen 0 und `RAND_MAX`.
`void srand(unsigned s);`	Initialisiert den Zufallsgenerator mit dem Wert `s`.

Tabelle 7.62 Mathematische Funktionen der Header-Datei <cstdlib> (Forts.)

`div_t` und `ldiv_t` sind Strukturtypen mit folgendem Inhalt:

```
typedef struct{
   int quot;  /* quotient */
   int rem;   /* remainder */
} div_t;
```

... bzw. ...

```
typedef struct{
   long int quot;  /* quotient */
   long int rem;   /* remainder */
} ldiv_t;
```

7.3.4 Grenzwerte von Zahlentypen

Ein Problem, mit dem sich jeder Entwickler auseinandersetzen muss, ist die plattformabhängige Darstellung von numerischen Werten. Zum Beispiel kann auf einem System ein Integer nur 16 Bit belegen, auf einem anderen 32 und wieder auf einem anderen 64 Bit. Um zur Laufzeit entsprechende Werte zu numerischen Werten zu ermitteln, steht Ihnen in der Header-Datei <*limits*> die Template-Klasse `numeric_limits` zur Verfügung. Diese Klasse liegt im Namensraum `std` und muss somit ebenfalls mit angegeben werden.

In dieser Template-Klasse sind alle Spezialisierungen für die ganzzahligen Datentypen `bool`, `char`, `wchar_t`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long` (eventuell `long long` und `unsigned long long`) sowie für die Gleitkommatypen `float`, `double`, `long double` enthalten.

[»] **Hinweis**

Für nicht fundamentale Typen wie beispielsweise `complex<T>` darf keine solche Spezialisierung vorgenommen werden.

Für die Spezialisierung dieser Grunddatentypen sind verschiedene von der Implementierung abhängige `public`-Funktionen und `public`-Eigenschaften definiert, die in den folgenden Tabellen (7.63 und 7.64) aufgelistet sind.

Eigenschaft	Beschreibung
`static const bool is_specialized;`	Gibt `true` zurück, wenn es eine Spezialisierung für diesen Datentyp in <*limits*> gibt, ansonsten `false`.
`static const int digits;`	Gibt die Anzahl Bits (ohne Vorzeichen) zurück, womit dieser Typ intern dargestellt wird. Bei Gleitpunktzahlen wird die Anzahl der Bits zurückgegeben, die zur Darstellung der Mantisse benötigt werden.
`static const int digits10;`	Gibt die Genauigkeit in Anzahl Dezimalziffer zurück (für `float` beispielsweise 6).
`static const bool is_signed;`	Gibt `true` zurück, wenn der Datentyp mit einem Vorzeichen interpretiert wird, ansonsten `false`.
`static const bool is_integer;`	Gibt `true` zurück, wenn der Datentyp ganzzahlig ist.
`static const bool is_exact;`	Gibt `true` zurück, wenn eine exakte Darstellung erlaubt ist (beispielsweise ganze und rationale Zahlen).
`static const int radix;`	Gibt bei Gleitpunktzahlen die Basis der internen exponentiellen Darstellung zurück. Bei Ganzzahlen wird die Basis der internen Darstellung zurückgegeben.
`static const int min_exponent;`	Gibt den kleinsten negativen ganzzahligen Exponent `ex` zur Basis `radix` zurück, so dass die Gleitpunktzahl $radix^{ex}$ noch darstellbar ist.
`static const int min_exponent10;`	Gibt den kleinsten negativen ganzzahligen Exponent `ex` zur Basis `10` zurück, so dass die Gleitpunktzahl 10^{ex} noch darstellbar ist.
`static const int max_exponent;`	Gibt den größten positiven ganzzahligen Exponent `ex` zur Basis `radix` zurück, so dass die Gleitpunktzahl $radix^{ex}$ noch darstellbar ist.
`static const int max_exponent10;`	Gibt den größten positiven ganzzahligen Exponent `ex` zur Basis `10` zurück, so dass die Gleitpunktzahl 10^{ex} noch darstellbar ist.
`static const bool has_infinity;`	Gibt `true` zurück, wenn der Datentyp zur Darstellung einen unendlich großen Wert zulässt.
`static const bool has_quiet_NaN;`	Gibt `true` zurück, wenn der Datentyp den Wert `NaN` (*Not a Number* = keine Zahl) ohne Signalisierung darstellen kann
`static const bool has_signaling_NaN;`	Gibt `true` zurück, wenn der Datentyp den Wert `NaN` (*Not a Number*) signalisieren kann.

Tabelle 7.63 »public«-Eigenschaften (Datenelemente) in »limits«

Eigenschaft	Beschreibung
`static const bool has_denorm;`	Gibt `true` zurück, wenn der Datentyp eine variable Anzahl Bits zur Darstellung des Exponenten erlaubt.
`static const bool has_denorm_loss;`	Gibt `true` zurück, wenn ein Verlust der exakten Darstellung durch die Denormalisierung und nicht durch ein ungenaues Rechenergebnis entsteht.
`static const bool is_iec559;`	Gibt `true` zurück, wenn der Datentyp der IEC(*International Electrotechnical Commission*)-559-Norm entspricht.
`static const bool is_bounded;`	Gibt `true` zurück, wenn die Menge der darstellbaren Werte des Typs endlich ist. `false` wird zurückgegeben, wenn es sich um einen Typ mit beliebiger Genauigkeit handelt.
`static const bool is_modulo;`	Gibt `true` zurück, wenn es sich um einen Modulo-Typ handelt. Ganzzahlige Typen ohne Vorzeichen beispielsweise sind Modulo-Typen. Gleitpunktzahlen sind keine Modulo-Typen.
`static const bool traps;`	Gibt `true` zurück, wenn für den Datentyp ein Trapping implementiert ist.
`static const bool tinyness_before;`	Gibt `true` zurück, wenn ein sehr kleines Ergebnis vor dem Runden erkannt wird.

`static const float_round_style round_style;`	Gibt die für den Datentyp verwendete Rundungsart zurück. Wird gewöhnlich nur für Gleitpunkttypen verwendet. Der Datentyp `float_round_style` ist ein `enum` mit folgenden möglichen Werten:
	`round_indeterminate = -1`
	`round_toward_zero = 0`
	`round_to_nearest = 1`
	`round_to_infinity = 2`
	`round_toward_neg_infinity = 3`

Tabelle 7.63 »public«-Eigenschaften (Datenelemente) in »limits« (Forts.)

Neben den Eigenschaften gibt es auch einige `public`-Methoden in *<limits>*. In der folgenden Tabelle (7.64) steht `T` für den entsprechenden spezialisierten Datentyp.

Methode	Beschreibung
`static T min() throw();`	Gibt den kleinsten darstellbaren Wert für `T` zurück.
`static T max() throw();`	Gibt den größten darstellbaren Wert für `T` zurück.

Tabelle 7.64 »public«-Methoden der Klasse »numeric_limits«

Methode	Beschreibung
`static T epsilon() throw();`	Gibt die kleinste positive Gleitpunktzahl zurück.
`static T round_error();`	Gibt ein Maß für den größten Rundungsfehler (nach ISO/IEC 10967 – 1, Teil 1, Abschnitt 5.2.8) zurück.
`static T quiet_NaN() throw();`	Darstellung des Wertes für `NaN` (*Not a Number*) ohne Signal
`static T signaling_NaN() throw;`	Darstellung des Wertes für `NaN` (*Not a Number*) mit Signal
`static T denorm_min() throw();`	Gibt den kleinsten positiven denormalisierten Wert zurück.

Tabelle 7.64 »public«-Methoden der Klasse »numeric_limits« (Forts.)

Die Anwendung dieser einzelnen Eigenschaften und Methoden auf spezielle Datentypen ist in der Praxis recht einfach. Die Syntax (`T` durch entsprechenden Datentyp ersetzen) lautet:

```
#include <limits>
using namespace std;
...
// Eigenschaft abfragen
numeric_limits<T>::eigenschaft;
// Methode verwenden
numeric_limits<T>::methode();
```

Hierzu ein Listing, das einige dieser Spezialisierungen in der Praxis zeigen soll:

```
// limits1.cpp
#include <limits>
#include <iostream>
using namespace std;

int main(void) {
   cout << "Anzahl Bits für ..." << endl;
   cout << "...char : "
        << numeric_limits<char>::digits << endl;
   cout << "...int  : "
        << numeric_limits<int>::digits << endl;
   cout << "...bool : "
        << numeric_limits<bool>::digits << endl;

   cout << "Genauigkeit in Dezimalziffern ..." << endl;
   cout << "...float : "
        << numeric_limits<float>::digits10 << endl;
```

```
   cout << "...double: "

        << numeric_limits<double>::digits10 << endl;

   cout << "Datentyp ..." << endl;
   cout << "...int ist "
        << ( numeric_limits<int>::is_signed
           ? "signed" : "unsigned" ) << endl;
   cout << "...unsigned int ist "
        << ( numeric_limits<unsigned int>::is_signed
           ? "signed" : "unsigned" ) << endl;

   cout << "Limits für Datentypen ..." << endl;
   cout << "int" << endl;
   cout << "min(): " << numeric_limits<int>::min() << endl;
   cout << "max(): " << numeric_limits<int>::max() << endl;

   cout << "double" << endl;
   cout << "min(): "
        << numeric_limits<double>::min() << endl;
   cout << "max(): "
        << numeric_limits<double>::max() << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Anzahl Bits für ...
...char : 7
...int  : 31
...bool : 1
Genauigkeit in Dezimalziffern ...
...float : 6
...double: 15
Datentyp ...
...int ist signed
...unsigned int ist unsigned
Limits für Datentypen ...
int
min(): -2147483648
max(): 2147483647
double
min(): 2.22507e-308
max(): 1.79769e+308
```

<climits> und <cfloat> (alias <limits.h> und <float.h>)

Die von der C-Programmierung her bekannten Definitionen in `<climits>` und `<cfloat>` bleiben selbstverständlich weiterhin erhalten. Ihre Benutzung wird aber in der Praxis nicht mehr empfohlen und ist auch nicht nötig, da alle Spezialisierungen mit den bereits gezeigten Eigenschaften und Methoden der Klasse `numeric_limits` realisiert werden können.

7.3.5 Halbnumerische Algorithmen

Bei den gleich folgenden allgemeinen numerischen Operationen handelt es sich um STL-Methoden auf Containern. Alle diese Algorithmen sind in der Header-Datei <*numeric*> enthalten, daher muss diese auch eingebunden werden.

Algorithmus	Beschreibung

`template<class InputIterator, class T> T accumulate( InputIterator first, InputIterator last, T init);`
 	
`template<class InputIterator, class T, class binaryOperation> T accumulate( InputIterator first, InputIterator last, T init, binaryOperation binOp);`
	Damit addieren Sie auf einen Startwert alle Werte `*i` eines Iterators `i` von `first` bis `last`. Wollen Sie statt einer Addition eine andere Operation verwenden, so existiert auch eine überladene Variante von `accumulate()`. Dabei wird die Operation als letzter Parameter übergeben. Hierfür können Sie auch die vordefinierten Funktoren verwenden – siehe Abschnitt 5.3.2, »Hilfsmittel (Hilfsstrukturen)«.

`template<class InputIterator1, class InputIterator2, class T> T inner_product( InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, T init);`
 	
`template<class InputIterator1, class InputIterator2, class T, class binaryOperation1, class binaryOperation2>`
	Damit wird das Skalar-Produkt zweier Container `u` und `v` addiert. Auch diese Version ist in einer überladenen Version vorhanden, um auch andere Operationen zu verwenden.

Tabelle 7.65 Numerische Algorithmen in der Header-Datei <numeric>

Algorithmus	Beschreibung

`T inner_product(`
`  InputIterator1 first1,`
`  InputIterator1 last1,`
`  InputIterator2 first2,`
`  T init,`
`  binaryOperation1 binOp1,`
`  binaryOperation2 binOp2);`
	(siehe Seite 811)

`template<class InputIterator,`
`  class OutputIterator>`
`OutputIterator partial_sum(`
`  InputIterator first,`
`  InputIterator last,`
`  OutputIterator result);`
 	
`template<class InputIterator,`
`  class OutputIterator,`
`  class binaryOperation>`
`OutputIterator partial_sum(`
`  InputIterator first,`
`  InputIterator last,`
`  OutputIterator result,`
`  binaryOperation binOp);`
	Dieser Algorithmus zur Partialsummenbildung ist dem von `accumulate()` recht ähnlich, nur dass das Ergebnis nach jedem Schritt in einem Container abgelegt wird, der durch den Iterator `result` angegeben wird. Außerdem gibt es bei `partial_sum()` keinen `init`-Wert. Sofern auch hier eine andere Operation benötigt wird, gibt es hierfür eine überladene Version.

`template<class InputIterator,`
`  class OutputIterator>`
`OutputIterator`
`adjacent_difference(`
`  InputIterator first,`
`  InputIterator last,`
`  OutputIterator result);`
 	
`template<class InputIterator,`
`  class OutputIterator,`
`  class binaryOperation>`
`OutputIterator`
`adjacent_difference(`
`  InputIterator first,`
`  InputIterator last,`
`  OutputIterator result,`
`  binaryOperation binOp);`
	Damit berechnen Sie die Differenz zweier aufeinanderfolgender Elemente im Container `v` und schreiben das Ergebnis in einen anderen Container, auf den mit dem Iterator `result` verwiesen wird. Da es genau einen Differenzwert weniger als Elemente gibt, bleibt das erste Element erhalten. Natürlich lassen sich damit außer Differenzbildungen noch andere Operationen (beispielsweise Fibonacci-Zahlen) durchführen. Falls andere Operationen nötig sind, ist auch eine überladene Version vorhanden.

Tabelle 7.65 Numerische Algorithmen in der Header-Datei <numeric> (Forts.)

Hierzu ein Beispiel, das diese Algorithmen in der Praxis zeigt:

```
// numeric1.cpp
#include <numeric>
#include <cmath>
#include <vector>
```

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

int main(void) {
   int dim = 4;
   vector<int> vec(5), vec2(5), res(5);
   vector<int> raum( dim, 1 );

   for( unsigned int i = 0; i < vec.size(); i++ ) {
      vec[i] = (i+1);
   }
   cout << "Summe aller Werte in vec : "
        << accumulate(vec.begin(), vec.end(), 0 )
        << endl;
   // ... oder die überladene Version mit dem
   // Funktor multiplies
   cout << "Produkt aller Werte in vec: "
        << accumulate( vec.begin(), vec.end(),
                       1L, multiplies<long>())
        << endl;

   // Das Beispiel berechnet die Länge eines
   // Vektors (1, 1, 1, 1) im R^4. Der Wert
   // für init muss 0 sein.
   cout << "Länge des Vektors : "
        << sqrt( (double) inner_product (
              raum.begin(), raum.end(), raum.begin(), 0 ))
        << endl;

   for( unsigned int i = 0; i < vec2.size(); i++ ) {
      vec2[i] = (i+1);
   }
   partial_sum( vec2.begin(), vec2.end(), res.begin() );
   cout << "Die Partialsummen von vec2 lauten: " << endl;
   for( unsigned int i = 0; i < res.size(); i++ ) {
      cout << res[i] << " ";
   }
   cout << endl;

   vector<int> vec3(5), vec4(5);
   for( unsigned int i = 0; i < vec3.size(); i++ ) {
      vec3[i] = i*i;
   }
   cout << "Die Differenzen: ";
```

```
   adjacent_difference(vec3.begin(), vec3.end(),
                       vec4.begin());
   for( unsigned int i = 0; i < vec4.size(); i++ ) {
      cout << vec4[i] << " ";
   }
   cout << endl;

   // Jetzt die Fibonacci-Zahlen
   vector<int> fibo(10);
   fibo[0] = 1; // Startwert
   cout << "Fibonacci-Zahlen (10) : ";
   adjacent_difference(fibo.begin(), fibo.end()-1,
                       (fibo.begin()+1), plus<int>());
   for( unsigned int i = 0; i < fibo.size(); i++ ) {
      cout << fibo[i] << " ";
   }
   cout << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Summe aller Werte in vec : 15
Produkt aller Werte in vec: 120
Länge des Vektors : 2
Die Partialsummen von vec2 lauten: 1 3 6 10 15
Die Differenzen: 0 1 3 5 7
Fibonacci-Zahlen (10) : 1 1 2 3 5 8 13 21 34 55
```

7.4 Typerkennung zur Laufzeit

Wenn Sie zur Laufzeit Informationen über einen Datentyp eines Objekts (oder auch eines gewöhnlichen Basisdatentyps) bekommen wollen, bietet Ihnen C++ Typinformationen zur Laufzeit an – auch *RTTI* (*Run Time Type Information*) genannt.

Vorwiegend werden diese Informationen bei polymorphen Klassen benötigt. Dazu wird vom Compiler für jede dieser Klassen eine Typinformation angelegt, und zwar als Objekt vom Typ `type_info`. Diese Informationen können Sie zur Laufzeit des Programms abfragen, um den aktuellen Typ des Objekts zu bestimmen. Die Klasse `type_info` ist wiederum in der Header-Datei <*typeinfo*> deklariert. In dieser sind auch die öffentlichen Methoden `==` und `!=` definiert, mit denen sich zwei Typen vergleichen lassen, sowie die Methode `name()`, die einen String über den Typnamen zurückliefert.

Um an Typinformationen im laufenden Programm zu kommen, wird der Operator `typeid()` verwendet. Die Syntax lautet:

```
typeid( ausdruck );
```

Als Argument können Sie hier entweder einen Typnamen oder einen beliebigen Ausdruck verwenden. Extraqualifizierer wie `const` und `volatile` werden nicht beachtet. Der `typeid()`-Operator liefert immer eine konstante Referenz auf das entsprechende Objekt von `type_info`. So ist ein Vergleich wahr, wenn `ival` vom Typ `int`, `const int` und `volatile int` ist:

```
if( typeid(ival) == typeid(int) )
```

Ein solcher Vergleich benötigt ein komfortableres Template wie beispielsweise:

```
template<class T1, class T2>
void check( T1& arg1, T2& arg2 ) {
   if( typeid(arg1) != typeid(arg2) ) {
      cout << "Fehler!!! Ungleiche Typen!!!" << endl;
      // Fehlerbehandlung hierhin. Beispielsweise Exception
   }
}
```

Hierzu ein komplettes Listing mit einfachen Datentypen:

```
// typeid1.cpp
#include <iostream>
#include <typeinfo>
using namespace std;

template<class T1, class T2>
void check( T1& arg1, T2& arg2 );

int main( void ) {
   int ival = 100;
   int ival2 = 111;
   float fval = 1.1;
   char cstring[] = "C String";
   string string1("C++ String");

   if( typeid(ival) == typeid(ival2) ) {
      cout << "ival und ival2 sind vom selben Typ" << endl;
   }
   else {
      cout << "ival und ival2 sind nicht vom selben Typ "
           << endl;
   }
```

```
   if( typeid(ival) == typeid(fval) ) {
      cout << "ival und fval sind vom selben Typ" << endl;
   }
   else {
      cout << "ival und fval sind nicht vom selben Typ"
           << endl;
   }
   if( typeid(ival) == typeid(int) ) {
      cout << "ival ist ein int" << endl;
   }
   check( cstring, string1 );
   check( ival, ival2 );
   return 0;
}

template<class T1, class T2>
void check( T1& arg1, T2& arg2 ) {
   if( typeid(arg1) != typeid(arg2) ) {
      cout << "Fehler!!! Ungleiche Typen!!!" << endl;
      // Fehlerbehandlung hierhin. Beispielsweise Exception
   }
}
```

Das Programm bei der Ausführung:

```
ival und ival2 sind vom selben Typ
ival und fval sind nicht vom selben Typ
ival ist ein int
Fehler!!! Ungleiche Typen!!!
```

Natürlich können Sie sich diese Typinformationen mit der Methode `name()` als »Klartext« zurückgeben lassen. Allerdings ist die Darstellung dieses C-Strings von der Implementierung abhängig. Hier dasselbe Beispiel nochmals mit der Methode `name()`:

```
// typeid2.cpp
#include <iostream>
#include <typeinfo>
using namespace std;

template<class T1, class T2>
void check( T1& arg1, T2& arg2 );

int main( void ) {
   int ival = 100;
   int ival2 = 111;
```

```
   float fval = 1.1;
   char cstring[] = "C String";
   string string1("C++ String");

   check( ival, fval );
   check( cstring, string1 );
   check( ival, ival2 );
   return 0;
}

template<class T1, class T2>
void check( T1& arg1, T2& arg2 ) {
   if( typeid(arg1) != typeid(arg2) ) {
      cout << "Fehler!!! Ungleiche Typen!!!" << endl;
      cout << typeid(arg1).name() << " != "
           << typeid(arg2).name() << endl;
      // Fehlerbehandlung hierhin. Beispielsweise Exception
   }
   else {
      cout << "Beide Typen sind gleich (Typ: "
           << typeid(T1).name() << ")" << endl;
   }
}
```

Das Programm bei der Ausführung:

```
Fehler!!! Ungleiche Typen!!!
i != f
Fehler!!! Ungleiche Typen!!!
A9_c != Ss
Beide Typen sind gleich (Typ: i)
```

Auf Objekte einer polymorphen Klasse liefert `typeid()` immer die Typinformation zum vollständigen Objekt zurück. Damit kann man sicherstellen, dass über Zeiger nur objektgemäße Funktionen aufgerufen werden. Im folgenden Beispiel wird überprüft, ob der Basisklasse-Zeiger auf ein `Vererbt`-Objekt zeigt oder nicht.

```
// typeid3.cpp
#include <iostream>
#include <typeinfo>
using namespace std;

class Basis {
public:
```

```
   virtual void vvfunc() {}
};

class Vererbt : public Basis {};

int main() {
   Vererbt* ptrV = new Vererbt;
   Basis*   ptrB = ptrV;
   if(  typeid( *ptrB ) == typeid( Basis ) ) {
      cout << "Es ist die Basisklasse" << endl;
   }
   else if( typeid( *ptrB ) == typeid( Vererbt ) ) {
      cout << "Es ist die vererbte Klasse" << endl;
   }
   cout << typeid( ptrB ).name() << endl;   // Basis*
   cout << typeid( *ptrB ).name() << endl;  // Vererbt
   cout << typeid( ptrV ).name() << endl;   // Vererbt*
   cout << typeid( *ptrV ).name() << endl;  // Vererbt
   delete ptrV;
   return 0;
}
```

Das Programm bei der Ausführung:

```
Es ist die vererbte Klasse
P5Basis
7Vererbt
P7Vererbt
7Vererbt
```

Wenn Sie den `typeid()`-Operator auf einen dereferenzierten `NULL`-Zeiger verwenden, wird eine Exception vom Typ `bad_typeid` ausgelöst (mehr zu den Exceptions und wie man diese abfängt in Kapitel 6, »Exception-Handling«).

[»] Hinweis

Informationen zu `static`-Typen lassen sich nur bei den eingebauten Datentypen einsetzen.

Mit der Methode `before()` können Sie die Beziehung zwischen Klassen bezüglich der Vererbungshierarchie überprüfen:

```
// typeid4.cpp
#include <iostream>
#include <typeinfo>
using namespace std;
```

```
class Basis {
public:
   virtual void vvfunc() {}
};

class Vererbt : public Basis {};

int main() {
   if( typeid(Basis).before(typeid(Vererbt)) ) {
      cout << "Basis ist die Basisklasse für Vererbt"
           << endl;
   }

   if(typeid(Vererbt).before(typeid(Basis)) ) {
      cout << "Vererbt ist die Basisklasse für Basis"
           << endl;
   }
   else {
      cout << "Vererbt ist nicht die Basisklasse für Basis"
           << endl;
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
Basis ist die Basisklasse für Vererbt
Vererbt ist nicht die Basisklasse für Basis
```

Zwar ist RTTI eine interessante Möglichkeit, Informationen zur Laufzeit zu ermitteln, aber man sollte diese Technik nur einsetzen, wenn es nicht mehr anders möglich ist, da RTTI für jeden im Programm vorkommenden Typ eine aufwendige Informationsstruktur anlegen muss. Bei vielen Compilern steht diese Option von RTTI nur mit speziellen Optionen zur Verfügung. Beispielsweise muss man beim GNU-Compiler die Option `-frtti` hinzufügen.

Als bessere Alternative kann man innerhalb von polymorphen Klassenhierarchien den Operator `dynamic_cast` verwenden. Mit diesem können Sie überprüfen, ob eine gewünschte Konvertierung zulässig ist oder nicht.

In diesem Kapitel möchte ich Ihnen ein paar Tipps, Hinweise und kurze Anleitungen mit auf den Weg geben für Probleme, auf die Sie früher oder später in Ihrer C++-Karriere garantiert stoßen werden. Da wäre beispielsweise das Entwickeln eigener Module, was für jeden C++-Programmierer von Anfang an von Bedeutung ist. Auch als Umsteiger von C auf C++ finden Sie hier wichtige Hinweise. Da es mittlerweile veraltetes C++ gibt, müssen auch hierzu einige Anmerkungen gemacht werden. Wie man umfangreichere Projekte (nicht nur in C++) effektiver planen, vorbereiten und vor allem auch visualisieren kann, wird Ihnen mit UML gezeigt. Natürlich soll, wie in jedem anderen Programmierbuch auch, der Programmierstil thematisiert werden. Außerdem wird noch auf eine Bibliothek von Boost (Boost.Regex) eingegangen – und nebenbei erfahren Sie auch noch, was es mit Boost auf sich hat.

8 Weiteres zum C++-Guru

8.1 Module

Wenn die Programme umfangreich werden, wird man nicht mehr mit einer Datei zum Kompilieren auskommen. In der Regel werden dann mehrere Quelldateien, auch *Module* genannt, verwendet.

Ein Modul ist im Grunde eine Sammlung von Klassen oder Funktionen. Im Laufe der Zeit werden Sie eine Menge Quellcodes schreiben. Dazu gehören Dinge wie zum Beispiel die Netzwerk- und Datenbankprogrammierung. Gewöhnlich packt man diese Klassen bzw. Funktionen in einzelne Module – eines für die Netzwerkprogrammierung und eines für die Datenbankprogrammierung. Dies allein hat schon den Vorteil, dass Sie dieses Modul immer wieder verwenden können und nicht abhängig vom Projekt sind bzw. die einzelnen Funktionalitäten erst suchen müssen. Und ganz nebenbei haben Sie Ihr Programm damit auch besser logisch strukturiert.

Allerdings gibt es neben der Wiederverwendung von Modulen noch weitere wichtige Gründe und Vorteile, den Quellcode in mehrere Module aufzuteilen. Gerade bei größeren Projekten werden Sie in der Regel recht selten allein arbeiten und die Arbeit zu mehreren Personen bewerkstelligen. Wenn hier alle Betei-

ligten mit ein und derselben Datei arbeiten müssten, wäre es wohl recht umständlich, die Änderungen einzubringen (was theoretisch mit einer Versionsverwaltungs-Software wie Subversion oder CVS möglich ist).

Das nächste Problem ist, dass zum Beispiel bei der Verwendung einer einzelnen Datei jedes Mal das komplette Programm neu übersetzt werden muss.

Leider gibt es keine festen Regeln, nach denen man einen Quellcode in einzelne Module aufteilt. Allerdings sollte man immer versuchen, ein Modul möglichst selbständig arbeiten zu lassen, der Austausch an Informationen mit anderen Modulen sollte möglichst gering ist. Und natürlich sollten die in einem Modul geschriebenen Funktionen bzw. Methoden alle eine recht ähnliche Aufgabe erfüllen.

8.1.1 Aufteilung

Zunächst werden Module gewöhnlich in zwei Teile aufgeteilt – in einen öffentlichen Teil und in einen versteckten. Mit dem öffentlichen Teil bieten Sie dem Benutzer die Schnittstelle an, damit er weiß, wie die Funktionen bzw. Klassen im Modul benutzt werden können. Gewöhnlich verwendet man für die öffentlichen Definitionen eine Header-Datei, die der Benutzer des Moduls einbinden muss. In dieser Header-Datei sind alle Deklarationen der Datenstrukturen, Funktionen, Klassen usw. enthalten – eben alles, was außerhalb des Moduls verwendet werden kann. In der Regel enden diese Header-Dateien mit *.h* bzw. *.hpp.*

Alles, was die Interna des Moduls betrifft, bzw. alles, was außerhalb des Moduls nicht verwendet werden soll – also der eigentliche, arbeitende Quellcode oder genauer gesagt: alles –, wird wiederum in eine eigene Datei gepackt. Gewöhnlich handelt es sich hierbei um eine Quelldatei mit der Endung *.cpp*, allerdings ohne eine `main()`-Funktion. Dabei muss diese Quelldatei nicht als Quellcode für den Benutzer des Moduls zur Verfügung stehen. Es reicht auch aus, wenn nur die Header-Datei(en) und die übersetzte Objektdatei (*.obj)* vorhanden sind – was durchaus gängige Praxis ist, wenn man den Quellcode nicht offenlegen will.

Allerdings haben besonders Anfänger häufig ein Problem mit dieser Vorgehensweise, was meistens zum Scheitern der Übersetzung führt. Wenn ein solches Beispiel mit Modulen auch funktionieren soll, muss die Header-Datei entweder im selben Verzeichnis sein wie der Quellcode des Hauptprogramms, oder man gibt das entsprechende Verzeichnis (mit `include`) an, in dem sich diese Datei befindet – es sei denn, die Header-Datei liegt im `include`-Header-Verzeichnis des Compilers. Außerdem ist es wichtig, beim Übersetzen des Quellcodes das Modul mit anzugeben (egal, ob dieses jetzt als Quelltext oder als Objektdatei vorhanden ist).

Hinweis [«]

Natürlich sollte man bedenken, dass die Objektdatei plattformspezifisch ist. Sollten Sie beispielsweise eine Objektdatei unter Linux erstellt haben, wird diese nicht unter MS Windows verwendbar sein.

Zum Schluss benötigen Sie selbstverständlich noch ein Hauptprogramm, das diese(s) Modul(e) verwendet. Fassen wir zusammen – für die Modularisierung benötigen Sie:

- Eine öffentliche Schnittstelle – eine Header-Datei (*.h* bzw. *.hpp*), in der sämtliche öffentlichen Deklarationen enthalten sind.
- Eine private Datei – im Allgemeinen eine einfache Quelldatei (*.cpp*) oder eine übersetzte Objektdatei (*.obj*), die beim Übersetzen dem Projekt hinzugefügt werden muss. In dieser Datei befindet sich die eigentliche »Arbeit« des Moduls.

Wir sprechen hier wohlgemerkt von *einem* Modul. In einem umfangreichen Projekt können viele solcher Module verwendet werden.

8.1.2 Die öffentliche Schnittstelle (Header-Datei)

Die Header-Datei ist gewöhnlich die Schnittstelle des Moduls und sollte laut Konvention mit *.h* enden (gelegentlich findet man auch *.hpp*). Eine solche Header-Datei enthält folgende Elemente:

- Kommentare, mit denen Sie dem Anwender mitteilen, was die ein oder andere Funktion bewirkt. Hierbei könnten auch weitere Informationen wie die letzte Änderung, Kopierrechte usw. stehen. Sofern Sie dem Anwender nur eine öffentliche Schnittstelle und eine Objektdatei (*.obj*) überlassen, ist eine Kommentierung unerlässlich.
- alle Schnittstellen von Klassen und Funktionen (Deklarationen)
- Konstanten
- Strukturen
- die `extern`-Deklarationen aller öffentlichen Variablen
- eine Präprozessor-Anweisung, die ein mehrfaches Inkludieren dieser Header-Datei verhindert, was generell immer zu Problemen führen kann. Dies wird mit dem folgenden einfachen Makro verhindert:

```
// wurde die Datei noch nicht eingebunden ...
#ifndef _DATEI_H
// ... dann binde diese jetzt ein ...
#define _DATEI_H
```

Als Makronamen können Sie eine beliebige Bezeichnung verwenden. Allerdings sollte man sinngemäß einen Namen wählen, den man mit dieser Datei in Verbindung bringt. Abgeschlossen wird diese Präprozessor-Anweisung am Ende der Datei durch:

```
#endif
```

Hier soll zu Demonstrationszwecken eine einfache Header-Datei namens *classA.h* erstellt werden:

```
// classA.h
// Mehrfaches Inkludieren verhindern
#ifndef _CLASSA_H_
#define _CLASSA_H_

class classA {
private:
   // Eigenschaften der Klasse classA
   int ival;
   float fval;

public:
   // Deklaration der Konstruktoren
   classA( int, float );
   classA( float, int );
   classA( int );
   classA( float );
   classA( );
   // Destruktor
   ~classA( ) {};

   // Fähigkeiten (Methoden) der Klasse class
   int getival( ) const;    // Gibt ival zurück
   float getfval( ) const;  // Gibt fval zurück
   void setival( int );     // Ändert ival
   void setfval( float );   // Ändert fval
   void printvals( ) const; // Gibt "(ival/fval)" aus
};
#endif
```

8.1.3 Die private Datei

In der privaten Datei befindet sich gewöhnlich die Implementierung der Schnittstelle aller Funktionen oder Methoden, die die eigentlichen Arbeiten ausführen. Neben dem arbeitenden Code findet man in der privaten Datei im Allgemeinen noch Folgendes:

- Kommentare – auch wenn Sie diese Datei nur als Objektdatei ausliefern wollen, sollten Sie es nicht versäumen, einige Kommentare einzubringen.
- die `include`-Anweisung(en), mit der (denen) Sie die öffentliche(n) Schnittstelle(n) der Module bzw. Bibliotheken mit einbinden.
- lokale Objekte, die nach außen hin nicht verfügbar sind. Zwar war es früher die Regel, solche Objekte als `static` zu deklarieren, aber in der Praxis stellt man solche Objekte mittlerweile in einen eigenen Namensraum.

Natürlich lassen sich in der privaten Datei noch weitere Klassen und Funktionen implementieren, die allerdings dann nicht mehr Teil der Schnittstelle sind, sondern gewöhnlich nur von den Funktionen bzw. Methoden der öffentlichen Schnittstelle verwendet werden.

Als Dateinamen einer solchen privaten Datei verwendet man meistens denselben wie bei der Header-Datei, nur jetzt mit der Endung *.cpp* – dies ist aber keine Regel.

Auch hierzu soll ein Beispiel erstellt werden, das die Arbeit für die öffentliche Schnittstelle *classA.cpp* übernimmt und implementiert:

```
// classA.cpp
// Enthält nur die Implementierung
// der öffentlichen Schnittstelle
#include <iostream>
#include "classA.h"
using namespace std;

// Die Konstruktoren
classA::classA( int i, float f ) {
   ival = i;
   fval = f;
}

classA::classA( float f, int i ) {
   ival = i;
   fval = f;
}

classA::classA( int i ) {
   ival = i;
   fval = 0.0;
}

classA::classA( float f ) {
   ival = 0;
```

```
   fval = f;
}

classA::classA( ) {
   ival = 0;
   fval = 0.0;
}

// Fähigkeiten (Methoden) der Klasse classA
int classA::getival( ) const {
   return ival;
}

float classA::getfval( ) const {
   return fval;
}

void classA::setival( int i ) {
   ival = i;
}

void classA::setfval( float f ) {
   fval = f;
}

void classA::printvals( ) const {
  cout << "(" << getival() << "/" << getfval()
       << ")" << endl;
}
```

8.1.4 Die Client-Datei

Die Client-Datei ist kein echter Fachbegriff, sondern wird nur von mir so verwendet. Sie ist die Datei, die auf das (oder die) Modul(e) zugreift – eben das Hauptprogramm mit einer `main()`-Funktion. Im Grunde müssen Sie nur noch die öffentliche Schnittstelle (Header-Datei) einbinden und gegebenenfalls die Datei *classA.cpp* zum Projekt hinzufügen. Hier eine solche Client-Datei, die auf das Modul `classA` zugreift:

```
// main.cpp
#include "classA.h"

int main() {
   classA cA;
   cA.printvals();
```

```
   cA.setival(99);
   cA.setfval(11.123);
   cA.printvals();
   return 0;
}
```

Das Programm bei der Ausführung:

```
(0/0)
(99/11.123)
```

8.1.5 Speicherklassen »extern« und »static«

Zwar wurden diese beiden Speicherklassen bereits beschrieben, doch passend zum Thema sollen sie hier nochmals aufgegriffen werden. Wie bereits im entsprechenden Abschnitt erwähnt, zeigen Sie mit dem Schlüsselwort `extern` an, dass eine Variable oder eine Funktion außerhalb der Datei definiert wurde. Ein einfaches Beispiel dazu:

```
// main.cpp
#include <iostream>
using namespace std;

// Externer Wert
extern int ival;
// Dupliziert den externen Wert
extern int dupl( );

int main() {
   cout << "ival = " << ival << endl;
   cout << "ival = " << dupl() << endl;
   cout << "ival = " << dupl() << endl;
   return 0;
}
```

Dieses Programm verwendet eine externe Variable `ival` und eine externe Funktion `dupl()`, die in einer anderen Datei definiert sind. Diese Datei müssen Sie natürlich beim Übersetzen mit angeben. In diesem Beispiel sieht die externe Datei wie folgt aus:

```
// externvars.cpp
int ival = 5;

int dupl( ) {
   return ival*=2;
}
```

Damit sieht die Ausgabe des Programms so aus:

```
ival = 5
ival = 10
ival = 20
```

Das Schlüsselwort `static` hat leider zwei Bedeutungen. Zum einen bedeutet es für global definierte Variablen oder Funktionen, dass sie nur für diese Datei gültig sind. Wenn in Funktionen Daten als `static` deklariert werden, bedeutet dies zum anderen, dass diese im statischen Speicher alloziert sind.

In der Praxis sollte man aber interne Objekte, die nur für diese Datei gültig sein sollen, nicht mehr mit `static` deklarieren, weil hierbei die Verwendung immer recht unklar ist. Für solche Zwecke sollte man stattdessen namenlose Namensräume verwenden – siehe Abschnitt 3.2.7, »Anonyme (namenlose) Namensbereiche«. Hierzu ein Beispiel, das den Zweck eines solchen namenlosen Namensraums zeigen soll:

```
// main.cpp
#include <iostream>
using namespace std;

int ival = 1;
float fval = 1.1;

int main() {
   cout << "ival = " << ival << endl;
   cout << "fval = " << fval << endl;
   return 0;
}
```

Obwohl der Code syntaktisch in Ordnung ist, gibt der Compiler eine Fehlermeldung aus. Hier gilt wieder die Regel, möglichst keine globalen Variablen zu verwenden. In unserem Projekt befindet sich nämlich folgende Datei:

```
// morevars.cpp
...
int ival = 5;
float fval = 5.5;
...
```

Um solche versteckten kleinen Fehler zu vermeiden, kann die entsprechende Variable mit dem Schlüsselwort `static` versehen werden oder – noch besser – in einem anonymen Namensraum definiert werden. Soll zum Beispiel die Variable der Datei *main.cpp* verwendet werden, können Sie Folgendes machen:

```
// main.cpp
#include <iostream>
using namespace std;

// Anonymer Namensraum, nur in dieser Datei gültig
namespace {
   int ival = 1;
   float fval = 1.1;
}

int main() {
   cout << "ival = " << ival << endl;
   cout << "fval = " << fval << endl;
   return 0;
}
```

Dasselbe hätten Sie auch erreicht, wenn Sie die Variablen in der Datei *morevars.cpp* anonymisiert hätten, nur mit dem Effekt, dass auf diese Variablen nur noch von der eigenen Datei zugegriffen werden kann. Hier ist also Vorsicht geboten, da man nicht weiß, ob andere Module ebenfalls auf diese Variablen zugreifen müssen.

```
// externvars.cpp

// anonymer Namensraum
namespace {
   int ival = 5;
   float fval = 5.5;
}
```

8.1.6 Werkzeuge

Natürlich ist hiermit das Thema »Entwickeln von Programmen« noch lange nicht erschöpft. Dazu sollte man sich spezielle Literatur anschaffen (oder eine entsprechende Dokumentation lesen), da diese Dinge meistens vom Compiler (oder der Entwicklungsumgebung) und der Plattform abhängig sind. Ich könnte Sie zwar mit Sätzen abspeisen wie »Hier wird das Tool XY beschrieben, das auf jedem System mit einem anderen Namen und einer anderen Syntax vorhanden ist«, aber wenn Sie auf einem anderen System oder mit einem anderen Compiler arbeiten, haben Sie davon nichts. Meistens verwendet man als Autor dann sowieso sein Lieblingswerkzeug.

Was ist dabei mit Tools oder Werkzeugen gemeint? Das fängt mit den Grundwerkzeugen eines jeden Programmierers an, dem Compiler und dem Linker, die häufig noch unzählige Optionen und Möglichkeiten anbieten.

Dann wären da noch Tools wie MAKE, die Ihnen das Übersetzen des Quellcodes abnehmen. Nicht nur das, denn es macht auch keinen Sinn, jedes Mal den kompletten Quellcode neu zu übersetzen, wenn sich nur eine Datei geändert hat. Auch hierbei hilft Ihnen ein Werkzeug wie MAKE. Außerdem ist es möglich, eigene Bibliotheken zu erstellen, wie diese bei Ihrem Compiler vorhanden sind.

Dann benötigt man häufig noch eine Versionsverwaltung, besonders wenn mehrere Entwickler an einem Projekt arbeiten. Mit solchen Programmen kann man erkennen, wer was gemacht und verändert hat. Hier gibt es freie Programme wie Subversion oder CVS.

Wollen Sie die Programme auf ihre Laufzeit messen und eventuell optimieren, benötigen Sie sogenannte Profiler. Solche Profiler unterscheiden sich erheblich voneinander. Einige messen nur die gesamte Laufzeit des Programms (was bei dauerhaft laufenden Programmen wenig Sinn macht), andere messen, wie lange eine Funktion oder gar wie oft eine Programmzeile ausgeführt wurde.

Ein häufig aus Zeit- und Kostengründen vernachlässigtes Thema ist das Überprüfen und Testen des Programms nach der Fertigstellung. Meistens sind die Kosten dann nach dem Ausliefern bedeutend höher, wenn der ein oder andere Bug wieder ausgebügelt werden muss. Zum Prüfen von Programmen gibt es sogenannte Debugger oder Programme, die Pufferüberläufe bzw. unerlaubte Speicherzugriffe aufspüren.

[»] **Hinweis**

Für Linux/UNIX-User finden Sie in meinem Buch »Linux-UNIX-Programmierung« (3. Auflage) ein ganzes Kapitel zu diesem Thema. Das Buch steht auch zum Onlinelesen auf meiner Webseite *www.pronix.de* zur Verfügung.

8.2 Von C zu C++

Häufig werden Programmierer beauftragt, ältere C-Programme in C++-Programme umzuschreiben. Hierbei treten immer wieder »Kompatibilitätsprobleme« auf. Daher finden Sie nun das Wichtigste dazu, wie man Probleme dabei vermeidet.

8.2.1 Notizen

- Einen echten C-Compiler können Sie erkennen, wenn dieser mit der Operation `sizeof('A')` die Größe eines `int` zurückgibt. Ein C++-Compiler gibt hierbei immer die Größe eines `char` zurück.
- Wurde in C++ eine `struct` deklariert, kann diese auch als Typname verwendet werden. C hingegen erwartet hier das Schlüsselwort `struct` vor jeder Deklaration:

```
struct foo { int ival; float fval };

foo foo_val;            // ok in C++, falsch in C
struct foo foo_val;     // ok in C und C++
```

- Ältere C-Compiler kennen keine `//`-Kommentare. Diese wurden erst in einem neueren Standard eingeführt.
- In C wurde häufig das Schlüsselwort `static` verwendet, um anzugeben, dass das Speicherobjekt nur lokal gültig ist. Für solche Zwecke wird in C++ mittlerweile empfohlen, anonyme Namensräume zu verwenden – siehe Abschnitt 3.2.7, »Anonyme (namenlose) Namensbereiche«.
- Casts im C-Stil sollte man durch Casts im neuen Stil (`static_cast`, `reinterpret_cast` und `const_cast`) ersetzen, wenn Typumwandlungen nötig sind (siehe Abschnitt 3.7.2, »Explizite Typumwandlung«).

8.2.2 Kein C++

In diesem Abschnitt finden Sie alles, was wirklich zu Problemen führen kann, wenn Sie es in einem C++-Projekt verwenden oder beibehalten wollen. Was hier beschrieben wird, ist reines C und kann nicht in C++ verwendet werden.

- In älterem C mussten keine Prototypen angegeben werden. Wurde zum Beispiel für eine Funktion namens `func` kein Prototyp angegeben, wurde implizit folgender Prototyp generiert:

```
// Standard-Prototyp bei keiner Deklaration
int func( );
```

  In C steht nämlich das `()` nicht für eine leere Argumentenliste wie in C++, sondern für eine Argumentenliste variabler Länge ohne Typprüfung (!) der Parameter. Es ist früher praktisch möglich gewesen, in C einen Prototyp wie folgt zu deklarieren:

```
int myfunc( );
...
int ival = myfunc( 5 );
```

Es erfolgte hier keine Überprüfung des Parametertyps. So war es sogar möglich, eine beliebige Anzahl von Argumenten zu übergeben. Das ist zwar ein schlechter Programmierstil, aber sofern Sie auf einen solchen Code stoßen sollten, wissen Sie, was zu tun ist. In C++ benötigen Sie hingegen immer Funktions-Prototypen.

[»] **Hinweis**

Sollten Sie wirklich auf solchen Code stoßen, müssen Sie nicht alle Funktions-Prototypen von Hand einsetzen, sondern Sie können sich ein Werkzeug wie *prototize* (GNU) ansehen.

- In älteren Programmen sind einige Funktionen noch im alten K&R-Stil – kein ANSI-C-Standard (!) – deklariert. Dies sieht wie folgt aus:

```
/* Funktionsdeklaration im alten K&R-Stil */
void func( a, b, c )
char *a;
int b;
float c;
{
   /* Funktionsrumpf */
}
```

  Solche Definitionen müssen umgeschrieben werden, weil in C++ Parametertypen und -namen in den Klammern der Funktionsdefinition geschrieben werden:

```
void func( char* a, int b, float c ) {
   // Funktionsrumpf
}
```

- Ebenfalls im älteren C (wieder kein ANSI-C-Standard mehr) gab es einen Standard-`int`-Datentyp, der automatisch verwendet wurde, wenn man beispielsweise Folgendes geschrieben hatte:

```
const ival = 10;  // Keine Typangabe, dann wurde int genommen

// ... Dasselbe bei Funktionen ohne Rückgabewerte
// ... keine Angabe beim Rückgabewert, dann Standard-int
func( int a, int b );
```

  Eine solche Verwendung ist kein C++ und mittlerweile auch kein C mehr. Aber wie erwähnt, es gibt immer noch viele ältere C-Programme.

- In C ist es auch möglich, ganzzahlige Werte an Variablen eines Aufzählungstyps (`enum`) zuzuweisen. Dies ist in C++ nicht möglich:

```
enum myBool { myfalse, mytrue };
myBool a = 1;   // Fehler in C++, Ok in C
```

- In C ist es erlaubt, dass ein `void*` auf der rechten Seite einer Zuweisung oder Initialisierung steht und auf der linken Seite ein beliebiger Typ. Ein Klassiker:

  ```
  // malloc gibt ein void* zurück
  int* ptr = malloc( n*sizeof(int));
  ```

  In C++ müssten Sie den Rückgabewert von `malloc()` in den Typ `int*` casten. Allerdings verwendet man in C++ ohnehin den Operator `new`, um Speicher zu reservieren.

8.2.3 Kein C

Wenn Sie ein C-Projekt pflegen oder erstellen müssen und in den letzten Jahren nur noch in C++ programmiert haben, dann gibt es viele Sprachmittel, die Sie hierbei nicht verwenden können. So sind die in Tabelle 8.1 aufgelisteten Schlüsselwörter reine C++-Schlüsselwörter und können nicht in C verwendet werden.

and	dynamic_cast	operator	true
and_eq	explicit	or	try
Asm	export	or_eq	typeid
bitand	false	private	typename
bitor	friend	protected	using
Bool	inline	public	virtual
catch	mutable	reinterpret_cast	wchar_t
class	namespace	static_cast	xor
compl	new	template	xor_eq
const_cast	not	this	
delete	not_eq	throw	

Tabelle 8.1 Reine C++-Schlüsselwörter

Bevor jetzt der Einwand kommt, dass C (zumindest mit Einführung des neuesten Standards) sehr wohl Dinge wie `and`, `and_eq` usw. kennt, sei angemerkt, dass es sich dabei »nur« um Makros in den Standard-Header-Dateien handelt. In der Praxis bedeutet dies, dass Sie diese Makros mit `#undef` »abschalten« können, wenn Sie die Schlüsselwörter von C++ verwenden wollen. In Tabelle 8.2 sind alle Makros von C aufgelistet, die in C++ Schlüsselwörter sind.

and	bitor	not_eq	wchar_t
and_eq	compl	or	xor
bitand	not	or_eq	xor_eq

Tabelle 8.2 C-Makros, die in C++ Schlüsselwörter sind

Jetzt noch eine Liste zu den wichtigsten Sprachmitteln, die (auch das neuere) C nicht kennt (wobei es sich im Grunde ja nur um eine Liste handelt, mit der C++ von C erweitert wurde):

- `const` in konstanten Ausdrücken
- Referenzen
- der boolesche Datentyp `bool`
- die neue Syntax zum Casten
- Klassen und alles, was damit zusammenhängt, wie zum Beispiel:
  - Methoden und Eigenschaften
  - Konstruktoren und Destruktoren
  - abgeleitete Klassen
  - virtuelle Methoden und abstrakte Klassen
  - Zugriffskontrolle in den Klassen (`public` / `protected` / `private`)
  - `friend`-Funktionen
  - Zeiger auf Elemente
  - statische und mutable Elemente
  - Operator-Überladung
- Templates
- Standardargumente
- Funktionen überladen
- Namensbereiche
- Scope-Operator
- echte Ausnahmebehandlung (`throw` / `catch`)
- Typidentifizierung zur Laufzeit

8.2.4 »malloc« und »free« oder« new« und »delete«

Es ist möglich, in Ihren C++-Programmen auch die C-Bibliotheksfunktionen `malloc()`/`free()` zu verwenden. Sie können `malloc()`/`free()` und `new`/`delete` sogar mischen (auch wenn das nicht unbedingt sauber erscheint).

Mal abgesehen davon, dass die Verwendung von `malloc()`/`free()` recht umständlich und fehleranfällig ist (ein echter C-Anhänger sieht das natürlich wieder anders), gibt es hierbei eine Falle, in die man tappen kann, wenn man die alten C-Funktionen zur Speicherverwaltung verwendet – nämlich, wenn Sie versuchen, Speicher für eine Klasse (`class`) zu reservieren. Die Funktion `malloc()` reserviert nur Speicher für diese Klasse, ruft aber keinen Konstruktor auf (was der

Operator `new` standardmäßig macht). Die Gefahr, dass diese Klasse damit falsch initialisiert wird, ist groß. Dasselbe Problem besteht beim Freigeben von Speicher mit `free()`. `free()` gibt zwar den Speicher wieder frei, ruft aber nicht den Destruktor der Klasse auf.

Sofern Sie also vorhaben, `malloc()`/`free()` für Klassen zu verwenden, ist unbedingt davon abzuraten; stattdessen sollte `new`/`delete` verwendet werden.

8.2.5 »setjmp« und »longjmp« oder »catch« und »throw«

In C hat man Ausnahmen (Exceptions) mit den Bibliotheksfunktionen `setjmp()` und `longjmp()` behandelt. Mit `setjmp()` wurde dabei eine beliebige Stelle markiert, die bei einer Ausnahme mit `longjmp()` angesprungen wurde. So gesehen bereitet dies keine Schwierigkeiten, und es würde sich auch ganz gut mit C++ vertragen, wenn hier nicht wieder das Problem mit den Klassen wäre. Wird beispielsweise ein `longjmp()` aus einer Funktion gemacht, die eine Klasse definiert und initialisiert hat, und die Kontrolle an die `setjmp()`-Funktion zurückgegeben, so wird nicht der Destruktor der Klasse aufgerufen, was normalerweise passieren würde, wenn der Fehler mit `catch` und `throw` behandelt würde. Die Funktion `longjmp()` ist eine C-Funktion, und C kennt eben keine Klassen bzw. Destruktoren. Daher der dringende Rat, beim Umschreiben von Programmen von C nach C++ alle `longjmp`/`setjmp` durch die Ausnahmebehandlung `catch`/`throw` zu ersetzen (siehe Kapitel 6, »Exception-Handling«).

8.3 »Altes« C++

So alt ist C++ eigentlich noch gar nicht, aber es gibt mittlerweile schon ein »veraltetes« C++. Veraltet heißt nicht unbedingt falsch, aber da Sie vielleicht auch den ein oder anderen Code gesehen haben, bei dem etwas anders gemacht wurde als in diesem Buch, sollte hier erwähnt werden, *warum* etwas anders gemacht wurde.

8.3.1 Header-Dateien mit und ohne Endung

Sicherlich mag man sich fragen, wo die traditionelle Header-Dateiendung *.h* der Standardbibliothek geblieben ist. In älteren C++-Implementierungen wurde diese noch verwendet – und aus Kompatibilitätsgründen kann sie auch heute noch verwendet werden.

Das Problem ist, dass mit der Zeit eine komplett neu definierte Version der Standardbibliothek entwickelt wurde (man kann sagen, die Standardbibliothek wurde verbessert). Leider konnte das Standardisierungskomitee jetzt nicht ein-

fach erklären, dass die alte Bibliothek nicht mehr zum Standard gehört. Somit wurde der Name zu einem Problem, das man schlicht dadurch gelöst hat, dass die Endung *.h* weggelassen wurde. Somit konnten alte Programme immer noch die alte Standardbibliothek verwenden.

Die Versionen der Standard-Header-Dateien unterscheiden sich zudem noch dadurch, dass die alten Versionen mit der Endung *.h* ihre Deklarationen noch alle im globalen Bereich definiert haben – die neuen Header-Dateien dagegen haben ihre Deklarationen mittlerweile alle im Namensbereich `std` gesetzt.

8.3.2 Standardbibliothek nicht komplett oder veraltet

Bei älteren C++-Implementierungen kam es manchmal vor, dass Teile der Standardbibliothek nicht vorhanden waren. Häufig waren oder sind dies Dinge wie die String-Bibliothek oder die C-Standardbibliothek. Natürlich können dies auch noch andere Bibliotheken sein. Unabhängig davon gibt es eigentlich keinen Grund, in C-Gewohnheiten zurückzufallen. Die STL-Bibliothek ist frei erhältlich und kann kostenlos aus dem Internet bezogen werden (beispielsweise über *http://www.sgi.com/tech/stl/*).

Auch werden Sie in älteren C++-Versionen kein `istringstream` oder `ostringstream` (aus *<sstream>*) vorfinden. Stattdessen wurden hierbei die Klassen `istrstream` und `ostrstream` aus *<strstream.h>* verwendet. Allerdings arbeiten diese Objekte noch auf C-Strings und weisen somit einige Mängel auf.

8.3.3 Namespaces (Namensbereiche)

In älteren C++-Implementierungen gab es noch keine Namensbereiche (`namespace`). Um hierbei Mehrdeutigkeiten zu vermeiden, sollte man wieder (entgegen der Aussage in Abschnitt 3.4.3, »Speicherklasse 'static'«) auf das Schlüsselwort `static` zurückgreifen. Vorsicht ist auch bei globalen Bezeichnern geboten, besonders bei häufig verwendeten Dingen wie zum Beispiel Strings. (Klassen-) Namen wie `String` oder `Bool` sollte man vermeiden, um diese nicht mit den echten Namen zu verwechseln. Hierzu bieten sich häufig Versionen mit einem Präfix an, beispielsweise `my_String` oder `my_Bool`.

8.3.4 Schleifenvariable von »for«

Ich musste einmal einen umfangreicheren älteren Quellcode übersetzen, der viele `for`-Schleifenkonstrukte in folgender Schreibweise enthielt:

```
{
   for( int i = 0; i < 10; i ++ ) {
```

```
        // ...
    }
    if( i % 2 ) {
          // ...
     }
}
```

Nach der älteren C++-Implementierung war es noch möglich, die Schleifenvariable `i` zu verwenden, da sich der Gültigkeitsbereich einer `for`-Schleifenvariablen bis zum Ende des Gültigkeitsbereiches erstreckt hat, wo die `for`-Anweisung definiert wurde. Solche Schleifenvariablen müssen Sie nach neuem Standard so schreiben, dass sie vor der `for`-Schleife deklariert werden.

8.4 UML

UML (*Unified Modeling Language*) ist eine formale Sprache, die zur Visualisierung, Konstruktion und Dokumentation von Softwaresystemen (und auch Geschäftsmodellen) verwendet wird. Damit ist es möglich, komplexere Zusammenhänge mit Hilfe objektorientierter Konzepte abzubilden.

Damit Sie mich nicht falsch verstehen: UML ist nicht abhängig von der Programmiersprache – UML unterstützt alle gängigen objektorientierten Programmiersprachen.

Hinweis [«]

UML unterstützt und verwendet viele Diagramme. Da Sie ein Buch zur C++-Programmierung in Händen halten, wird hier nur auf das Klassendiagramm eingegangen. Auch sollte man keine Referenz erwarten. Hierzu ist spezielle Literatur sehr sinnvoll. Zu empfehlen ist »UML 2, das umfassende Handbuch« von Christoph Kecher bei Galileo Press.

Hinweis [«]

Für das Verständnis des folgenden Abschnitts ist es wichtig, dass Sie die Konzepte der objektorientierten Programmierung (siehe Kapitel 4, »Objektorientierte Programmierung«) verstanden haben.

8.4.1 Wozu UML?

Eine Frage, die sich ein Programmierer wohl häufig stellt, ist die, wofür er das ganze »grafische Zeugs« mit den Diagrammen lernen soll. Dazu ein einfaches Beispiel: Sie müssen ein Programm für eine Firma erstellen, die Autos vermietet. Wie fangen Sie ein solches Projekt an? Viele angehende Programmierer program-

mieren hier einfach drauflos. Das ist zunächst nicht so schlimm, aber wie sieht es mit dem Kunden aus? Sind Sie sich sicher, dass Sie den Kunden richtig verstanden haben? Was ist, wenn Sie nach der Fertigstellung feststellen müssen, dass sich der Kunde das ganz anders vorgestellt hat?

Oder wie sieht es aus, wenn Sie in einem Entwicklungsteam arbeiten oder es sogar leiten? Die Aufgaben werden verteilt, aber wie kann man sicher sein, dass jeder Programmierer alles verstanden hat? Oder was ist, wenn der ein oder andere Programmierer eine Veränderung einbringen will oder bereits eingebracht hat?

Wie kann man also dem Kunden oder den Kollegen ein anschauliches und verständliches Gesamtbild vermitteln und die Masse an Informationen begreiflich machen? Hierzu bedarf es einer ganz besonderen Organisation, und genau dafür eignet sich UML hervorragend. UML kann theoretisch auch von demjenigen verstanden werden, der nicht damit vertraut ist (somit auch vom Kunden).

Aber so tief will ich jetzt gar nicht in die Materie einsteigen. Wir wollen eben nur unser Projekt für die Firma realisieren. Damit Sie einen ersten Eindruck von UML bekommen, will ich ein Beispiel zum Projekt abbilden, was natürlich nur ein erster Entwurf sein soll und ganz anders aussehen kann (je nach Bedarf).

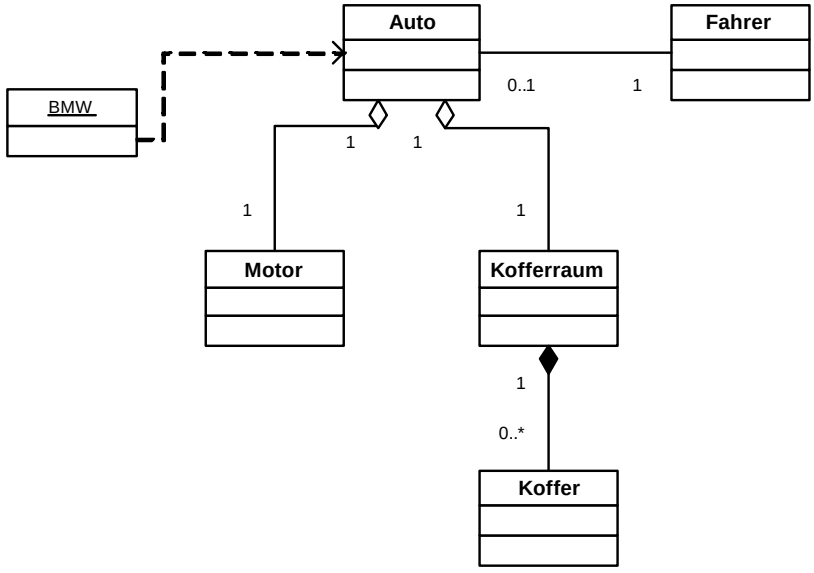

Abbildung 8.1 UML-Ansicht eines Autos

Das Diagramm der Abbildung 8.1 zeigt, dass hier zum Beispiel `BMW` ein Objekt der Klasse `Auto` ist (was durch den gestrichelten Pfeil dargestellt wird). Zu einem Auto gehört auch ein `Fahrer`, der das Auto mietet. In jedem Auto kann ein Fahrer fahren, während ein Fahrer nur in einem oder gar keinem Auto (`0..1`) sitzen kann. Die Klasse `Auto` besitzt einen `Motor`, und ein Motor gehört hier zu einem Auto (daher jeweils `1`). Außerdem hat ein Auto einen `Kofferraum`, der wiederum nur zu einem Auto gehört (wieder jeweils mit `1` kurz beschrieben). In diesem `Kofferraum` wiederum können keine oder viele `Koffer` (`0..*`) liegen, während die Koffer eben nur im Kofferraum liegen können (wer sagt da jetzt was vom Rücksitz).

Schon hätten Sie ein erstes Grundgerüst für Ihr Projekt. Das wird auf den folgenden Seiten deutlicher werden.

8.4.2 UML-Komponenten

Wie Sie im letzten Abschnitt erfahren konnten, besteht UML aus grafischen Elementen, die nach gewissen Regeln kombiniert ein Diagramm bilden. UML (ab Version 2.0, auch als *UML 2* bezeichnet) unterscheidet zwischen den *Strukturdiagrammen* und den *Verhaltensdiagrammen* (insgesamt gibt es derzeit 13 Diagrammarten).

Zu den (sechs) Strukturdiagrammen zählen:

- Klassendiagramm
- Kompositionsstrukturdiagramm (Montagediagramm)
- Komponentendiagramm
- Verteilungsdiagramm
- Objektdiagramm
- Paketdiagramm

Die (sieben) Verhaltensdiagramme sind:

- Aktivitätsdiagramm
- Sequenzdiagramm
- Kommunikationsdiagramm
- Interaktionsübersichtsdiagramm
- Zeitverlaufsdiagramm
- Anwendungsfalldiagramm (Nutzfalldiagramm)
- Zustandsdiagramm

[»] **Hinweis**

Auf die einzelnen Diagramme einzugehen würde den Rahmen dieses Buches sprengen. Aber um den Bezug zum Thema (C++ und OOP) nicht zu verlieren, werden wir uns vorwiegend mit dem Klassen- und dem Objektdiagramm begnügen (was vorerst auch ausreichend ist).

8.4.3 Diagramme erstellen

Wie Sie UML-Diagramme erstellen, bleibt Ihnen überlassen. Sie können entweder zu Papier und Bleistift greifen, oder Sie verwenden eines der vielen vorhandenen Programme.

Die eine Gruppe der Programme hilf Ihnen beim Zeichnen von Diagrammen, ohne dass sie die Modellelemente, die den grafischen Elementen auf den Diagrammen entsprechen, in einem Repository ablegen. Als Beispiel könnte ich hier entweder Microsofts Visio (kommerziell) oder Dia (Open Source) nennen. Wobei Visio nur unter MS-Systemen vorhanden ist. Die andere Gruppe der Programme unterstützt die Erstellung von Modellen und das Zeichnen von Diagrammen, die UML unterstützt (siehe Abschnitt 8.4.2, »UML-Komponenten«). Hier könnte ich das Programm Rational Software Architect (kommerziell) oder ArgoUML (frei) empfehlen.

8.4.4 Klassendiagramme mit UML

Auf den folgenden Seiten will ich Ihnen UML anhand der Objektorientierung näherbringen. Was OOP ist, haben Sie ja bereits ausführlich in Kapitel 4, »Objektorientierte Programmierung«, erfahren.

Die Klasse

Eine Klasse wird in UML durch ein rechteckiges Symbol angezeigt. Nach der Konvention (kein Standard) sollte der Klassenname mit einem Großbuchstaben beginnen. Beispielsweise wird eine Klasse `Buch` wie

```
class Buch {
   // ...
};
```

mit folgendem Klassensymbol in UML dargestellt:

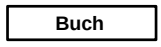

Abbildung 8.2 Klassensymbol von UML

Die Attribute (Eigenschaften) einer Klasse

In UML sind die Attribute einer Klasse das, was Sie als Programmierer die Eigenschaften (bzw. Daten) nennen. Diese Attribute werden getrennt durch eine Linie unterhalb des Klassennamens (im Klassensymbol) geschrieben:

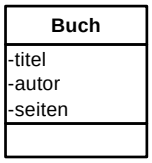

Abbildung 8.3 Eine Klasse mit ihren Attributen

Auf einen C++-Code angewandt, sieht dies (siehe Abbildung 8.3) folgendermaßen aus:

```
class Buch {
   // Eigenschaften (Attribute) der Klasse:
   private:
   char titel[50];
   char autor[50];
   unsigned int seiten;
};
```

In der Praxis sollte man für jedes Attribut einen bestimmten Wertebereich festlegen. Gewöhnlich entspricht eine solche Festlegung dem Datentyp einer Programmiersprache. Solche Attribute lassen sich mit den klassischen Datentypen einschränken. Mit den Datentypen in C++ würden die Attribute der Klasse wie folgt aussehen:

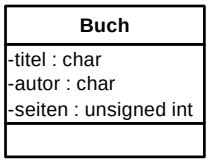

Abbildung 8.4 Der Wertebereich der Attribute einer Klasse wurde festgelegt.

In diesem Buch wird zwar C++ verwendet, aber es kann durchaus sein, dass Sie – das ist häufiger der Fall – folgende Notation des Wertebereichs finden:

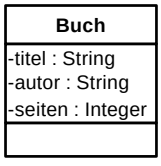

Abbildung 8.5 Andere Notation des Wertebereichs

Visual-Basic-Anhängern wird diese Schreibweise geläufig sein. Aber ich denke, man kann relativ gut erkennen, welche Typen hier gemeint sind (siehe auch Tabelle 8.3). Diese Typen sind in der OCL (*Object Constraint Language*) festgelegt.

[»] **Hinweis**

Die OCL ist Bestandteil von UML und dient unter anderem der textuellen Spezifikation von Invarianten in Klassendiagrammen, von Bedingungen in Sequenzdiagrammen oder der Formulierung von Vor- und Nachbedingungen für Methoden. Ihre Syntax ist an die Programmiersprache Smalltalk angelehnt. Sie ist seit der UML-Version 1.1 Bestandteil von UML.

Typ	Bedeutet in C++	Beschreibung
`Char`	`char`	Zeichen
`Integer`	`int`, `unsigned int`, `long`, `unsigned long` etc.	Ganzzahlen
`Real`	`float`, `double`, `long double`	Gleitpunktzahlen
`String`	`char*`, `string`	Zeichenkette
`Boolean`	`bool`	Wahrheitswert

Tabelle 8.3 Gängige Angabe von Wertebereichen bei den Attributen

Zu den Attributen können auch sogenannte *Initialwerte* festgelegt werden. Wollen Sie zum Beispiel die Eigenschaft `seiten` mit dem Wert `0` vorbelegen, sieht dies so aus:

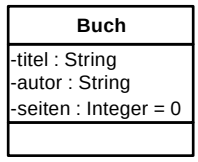

Abbildung 8.6 Ein Attribut mit einem Initialwert vorbelegen

Sobald ein Objekt der Klasse `Buch` angelegt wird, ist somit der Integer `seiten` mit dem Wert `0` vorbelegt. Wollen Sie hingegen eine Zusicherung anzeigen, dass der Wert des Attributs `seiten` immer größer als 0 sein muss bzw. ist, dann kann dies wie folgt visualisiert werden:

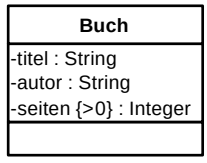

Abbildung 8.7 Ein Attribut mit der Zusicherung eines Wertes

Das Objekt einer Klasse

Ein Objekt wird – wie seine »Schablone«, die Klasse – ebenfalls mit einem rechteckigen Symbol gezeichnet. Der Unterschied zur Klasse besteht darin, dass der Objektname unterstrichen ist. Ein solches Objekt einer Klasse kann auf zwei unterschiedliche Arten kenntlich gemacht werden. Die erste Möglichkeit besteht darin, einen gestrichelten Pfeil zu verwenden, der vom Objekt ausgehend auf die Klasse zeigt (siehe Abbildung 8.8).

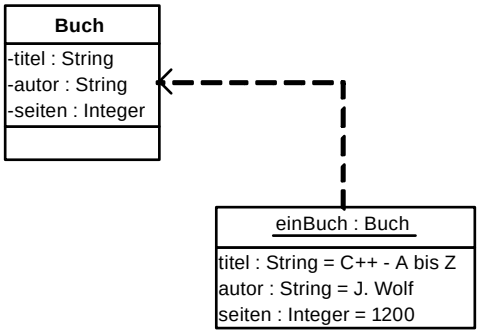

Abbildung 8.8 Ein Objekt einer Klasse

Die zweite Möglichkeit ist, nur den Klassennamen zu nennen. Allerdings muss dieser mit einem Doppelpunkt eingeleitet werden.

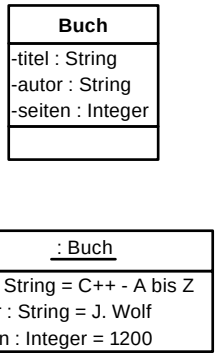

Abbildung 8.9 Noch eine Möglichkeit, ein Objekt einer Klasse darzustellen

Logischerweise finden Sie auch hier die in der Klasse definierten Attribute im Objekt wieder. Im Beispiel (siehe Abbildungen 8.8 und 8.9) sind diese sogar mit Werten belegt.

Die Methoden (in UML auch *Operationen*) einer Klasse werden im Objekt nicht genannt, weil diese ja innerhalb einer Klasse definiert sind und somit für alle Objekte gleich sind.

Die Operationen (Methoden) einer Klasse

Die Operationen (Fähigkeiten, Methoden oder meinetwegen auch Funktionalität) der Klassen werden unterhalb der Attribute, getrennt durch eine Linie, geschrieben. Hierzu das Diagramm zur Klasse `Buch`, das nun mit einigen Operationen ausgestattet wurde (siehe Abbildung 8.10).

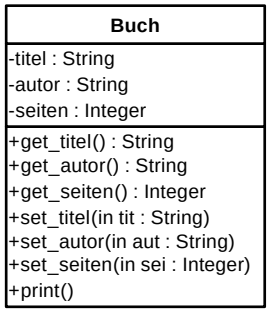

Abbildung 8.10 Die Liste mit allen Operationen der Klasse (mit Signatur)

Diese Angaben lassen sich recht einfach beschreiben. Eine Zeile wie beispielsweise

```
get_titel() : String
```

bedeutet, dass diese Operation mit dem Namen `get_titel` keine Argumente (leere Klammern) erhält, aber einen »String« zurückgibt. Hingegen bedeutet die Angabe

```
set_titel( tit : String )
```

dass diese Operation keinen Wert zurückgibt, aber als Argument den Typ `String` mit dem Bezeichner `tit` erwartet. Mehrere Argumente werden durch ein Komma getrennt:

```
set_titel_und_seiten ( tit : String, sei : Integer )
```

Die Operation `set_titel_und_seiten` erwartet zwei Argumente, ein Argument vom Typ `String` mit der Bezeichnung `tit` und ein Argument vom Typ `Integer` mit der Bezeichnung `sei`. Gibt diese Methode etwas zurück, zum Beispiel einen Integer, würde dies so aussehen:

```
set_titel_und_seiten(tit : String, sei : Integer) : Integer
```

Hinweis [«]

Bei den Attributtypen, Rückgabewerten und Argumenten sollten Sie immer bedenken, wem Sie das Diagramm vorlegen. Einen Laien (was der Kunde meistens ist) werden Sie mit der Angabe der Datentypen wohl eher abschrecken. Bei der Projektplanung allerdings sollten Sie sich Gedanken machen, welchen Typ Sie verwenden wollen. Dies hilft sehr beim Übergang vom UML-Modell zum C++-Projekt.

Angewendet auf einen C++-Code sieht die Klasse `Buch` aus Abbildung 8.10 jetzt wie folgt aus:

```
class Buch {
private:
   char titel[50];
   char autor[50];
   unsigned int seiten;

public:
   Buch( const char* titel="",
         const char* autor="", unsigned int seiten=0 ) {
      strncpy( this->titel, titel, 50 );
      strncpy( this->autor, autor, 50 );
      this->seiten = seiten;
   }
   ~Buch() {}
   const char* get_titel() const { return titel; }
   const char* get_autor() const { return autor; }
   unsigned int get_seiten() const { return seiten; }
   void set_titel( const char* tit ) {
      strncpy( titel, tit, 50 );
   }
   void set_autor( const char* aut ) {
      strncpy( autor, aut, 50 );
   }
   void set_seiten( unsigned int sei ) { seiten = sei; }
   void print() {
      cout << "Buchtitel : " << titel << "\n";
      cout << "Autor     : " << autor << "\n";
      cout << "Seiten    : " << seiten << "\n";
   }
};
```

Sichtbarkeit

Sicherlich haben Sie sich schon gefragt, was es bei UML-Diagrammen mit + oder – auf sich hat. Damit legen Sie die Sichtbarkeit der Attribute (Eigenschaften) und Operationen (Methoden) nach außen hin fest. Dass Eigenschaften gewöhnlich

`private` sein sollten und Methoden `public`, können Sie daraus schließen, dass ein – für `private` und ein + für `public` steht. Insgesamt werden in UML vier Sichtbarkeitsstufen definiert:

UML-Symbol	Sichtbarkeit	Beschreibung
+	`public`	öffentlich für alle Klassen sicht- und nutzbar
–	`private`	nur intern in der Klasse nutzbar; nach außen nicht sichtbar
#	`protected`	nur für die Klasse selbst und die Unterklassen sicht- und nutzbar
~	`package`	für die Klasse selbst und für alle Klassen desselben Pakets sichtbar und nutzbar

Tabelle 8.4 Sichtbarkeit von Attributen und Operationen

[»] **Hinweis**

Bitte beachten Sie, dass sich die Regeln der Sichtbarkeit in den einzelnen Programmiersprachen ein wenig unterscheiden können, auch wenn hierbei dieselben Schlüsselwörter verwendet werden.

Natürlich gilt auch für die Sichtbarkeitssymbole dasselbe wie für die Datentypen. Wenn Sie dem Kunden oder einem Anwender ein Analysemodell vorlegen wollen, sollten Sie dies nicht verwenden, weil es eher verwirrend oder störend wirkt. Für den Übergang vom UML-Diagramm zum Programmcode erleichtern einem diese Sichtbarkeiten aber wieder die Arbeit und sollten hierbei immer wieder verwendet werden.

Klassenattribut

In der Praxis wird man zwar die Attribute und Operationen in einer Klasse vereinbaren und von einem anschließend gebildeten Objekt festlegen (mit Werten initialisiert). Aber es ist auch möglich, ein Attribut nur als Klassenattribut zu definieren. Dann ist dieses Attribut nur über die Klasse erreichbar. Dies wurde bereits in der Praxis gezeigt, als eine Eigenschaft (Attribut) einer Klasse mit `static` deklariert wurde (siehe Abschnitt 4.4.15, »Statische Klassenmethoden«). Somit besitzt jedes erzeugte Objekt denselben identischen Wert, verweist also auf denselben Speicherbereich. Eine Änderung dieser Eigenschaft bedeutet, dass alle anderen Objekte auch davon betroffen sind. Natürlich ist dies auch mit Operationen möglich.

Solche Klassenoperationen und Klassenattribute werden in UML wie gewöhnlich notiert, nur werden diese zur Unterscheidung unterstrichen. Wenn Sie beispielsweise in der Klasse `Buch` ein Klassenattribut `verlag` hinzufügen, das mit dem Wert `Galileo` vorbelegt ist, sieht dies im UML-Diagramm aus, wie in Abbildung 8.11 dargestellt.

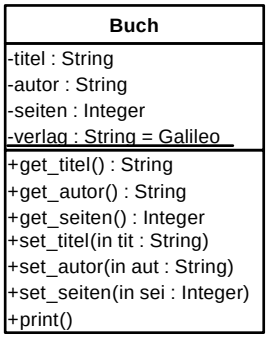

Abbildung 8.11 Klassenattribut

Damit ist sichergestellt, dass alle Bücher aus demselben Verlag kommen. Und sollte sich der Verlag ändern, dann geschieht dies nur über die Klasse und muss nicht für jedes Objekt vorgenommen werden.

Einiges zur Visualisierung von Klassen in der Praxis

Wenn Sie Klassen isoliert behandeln, wie dies bisher geschehen ist, zeigt man normalerweise alle Elemente einer Klasse an. Aber in der Praxis, wo Sie mehr als nur eine Klasse anzeigen wollen, ist dies nicht sehr übersichtlich. Ein solches Diagramm wirkt oft überladen. Häufig findet man daher Diagramme, in denen nur der Klassenname angegeben ist und die Felder für Operationen und/oder Attribute leer gelassen werden:

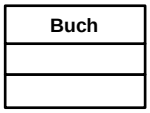

Abbildung 8.12 In der Regel zeigt man nicht alle Operationen und Attribute einer Klasse an.

Hierbei lässt sich allerdings auch ein Hinweis anbringen, so dass ein geübter »UML-Diagramm-Leser« weiß, dass einige Attribute und/oder Operationen ausgelassen wurden. Dazu verwendet man eine Ellipse (wird mit drei Punkten angegeben, vgl. Abbildung 8.13).

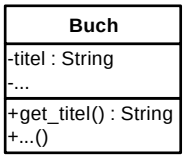

Abbildung 8.13 Die Verwendung einer Ellipse zeigt, dass nicht alles angegeben wurde.

Wird eine Liste recht lang, können Sie die Attribute und Operationen auch mit einem Stereotyp versehen. Dadurch lässt sich eine Klasse gut organisieren. Auch kann man damit praktisch einen Oberbegriff für eine Teilmenge von Attributen oder Operationen setzen. Ein solches Stereotyp wird in doppelten spitzen Klammern – auch als *Guillemets* bezeichnet – geschrieben: »Stereotyp« (siehe Abbildung 8.14).

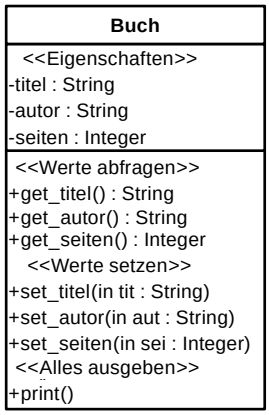

Abbildung 8.14 Die Klasse mit einem Stereotyp organisieren

[»] **Hinweis**

Ein Stereotyp ist vielseitig einsetzbar. Es kann auch oberhalb einer Klasse verwendet werden, um zu beschreiben, was die Klasse macht.

Notizen

Es ist auch möglich, den Attributen und Operationen eine Notiz hinzuzufügen, die entweder einen Attributwert oder eine Operation näher erläutert. Natürlich kann eine solche Notiz auch einen beliebigen Text enthalten (oder auch eine Grafik). In Abbildung 8.15 sehen Sie, wie man eine solche Notiz hinzufügt.

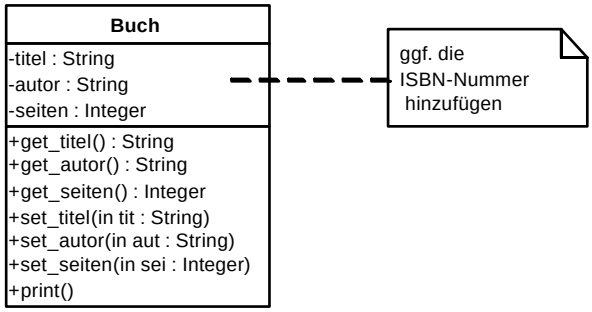

Abbildung 8.15 Beigefügte Notiz

Vererbung

Vererbte Klassen werden in UML durch einen Pfeil dargestellt, der immer von der Unterklasse auf die Oberklasse zeigt (siehe Abbildung 8.16).

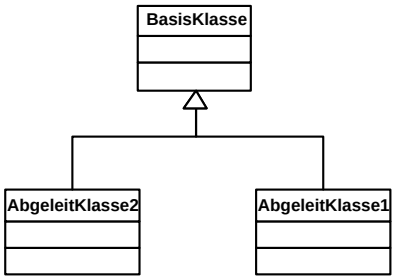

Abbildung 8.16 Darstellung einer Vererbung in UML

Sicherlich erinnern Sie sich noch an das erste Beispiel in der Praxis zur Vererbung (siehe Abschnitt 4.7.2, »Die Ableitung einer Klasse«), in dem Sie die Klasse `Buch` von der Basisklasse `Gegenstand` abgeleitet haben. Dies wollen wir uns jetzt auch in der UML-Darstellung ansehen. Somit haben Sie neben der Klasse `Buch` nun eine Klasse `Gegenstand`, die sich zusammen wie folgt in einem UML-Diagramm abbilden lassen:

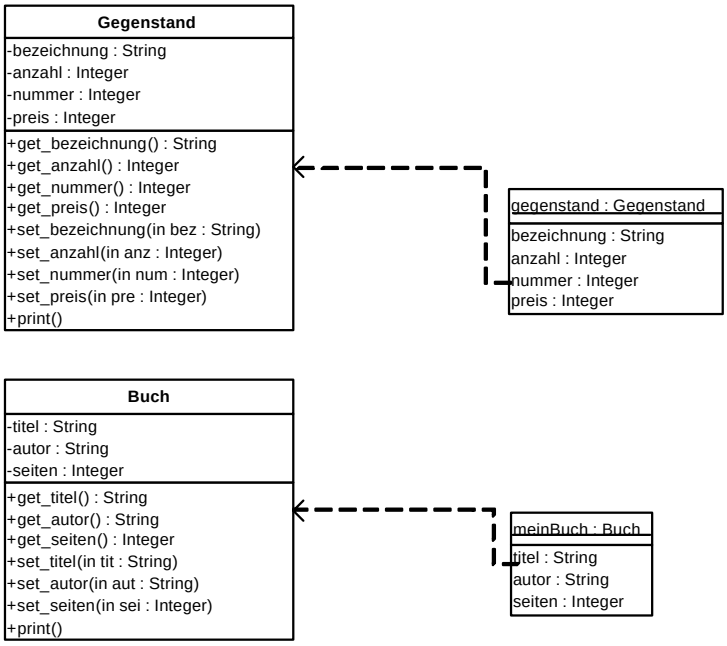

Abbildung 8.17 Zwei Klassen mit ihren Objekten mit UML dargestellt

Die Vererbung wurde codetechnisch wie folgt vorgenommen:

```
class Buch : public Gegenstand {
   // ...
};
```

Durch diese Vererbung ergibt sich im UML-Diagramm nun folgendes Bild:

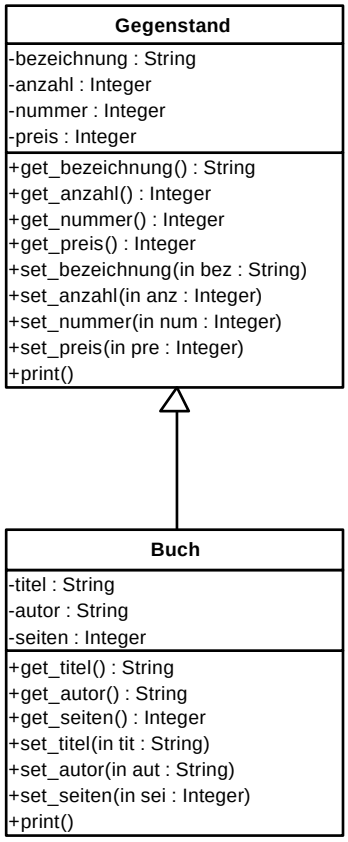

Abbildung 8.18 »Buch« wird jetzt von »Gegenstand« abgeleitet.

Wenn nun ein Objekt der Klasse `Buch` instantiiert wird, gehören auch die Attribute der Basisklasse `Gegenstand` dazu (siehe Abbildung 8.19). Somit stehen alle Attribute der Klasse `Gegenstand` jetzt auch für die Klasse `Buch` zur Verfügung – aber die Vererbung in C++ wurde ja bereits in Abschnitt 4.7, »Vererbung (abgeleitete Klassen)«, näher beschrieben.

[»] Hinweis

UML bezeichnet die Vererbung häufig auch als *Generalisierung*.

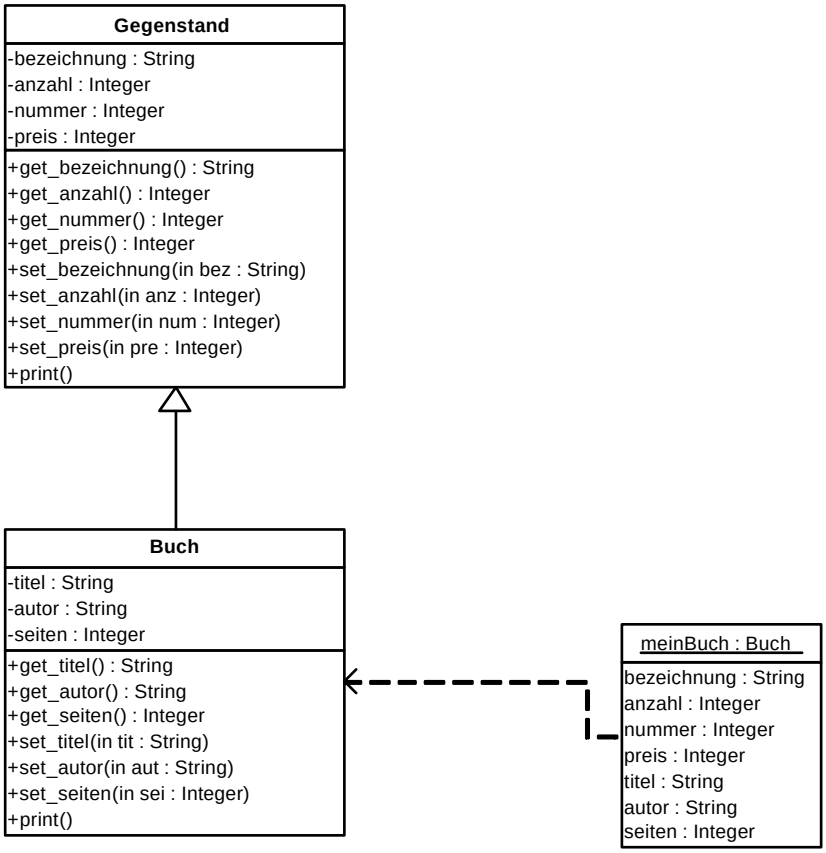

Abbildung 8.19 Ein Objekt der abgeleiteten Klasse »Buch« wird angelegt.

Mehrfachvererbung

Natürlich ist es auch möglich, in UML die Mehrfachvererbung darzustellen, so wie Sie dies in Abschnitt 4.9, »Mehrfachvererbung«, mit der Klasse `Wurstbrot` gemacht haben, wo von einem Objekt vom Typ `Wurstbrot` alle Datenelemente der Klassen `Wurst` und `Brot` abgeleitet wurden. Die Darstellung einer solchen Mehrfachvererbung – auf das Beispiel mit der Klasse `Wurstbrot` bezogen – wird entsprechend Abbildung 8.20 abgebildet.

In der Abbildung können Sie auch erkennen, dass die Attribute hier mit # als `protected` abgebildet wurden. Damit sind die Attribute weiterhin nach außen nicht sichtbar, dafür aber innerhalb der Klassen gültig und von `Wurstbrot` somit verwendbar.

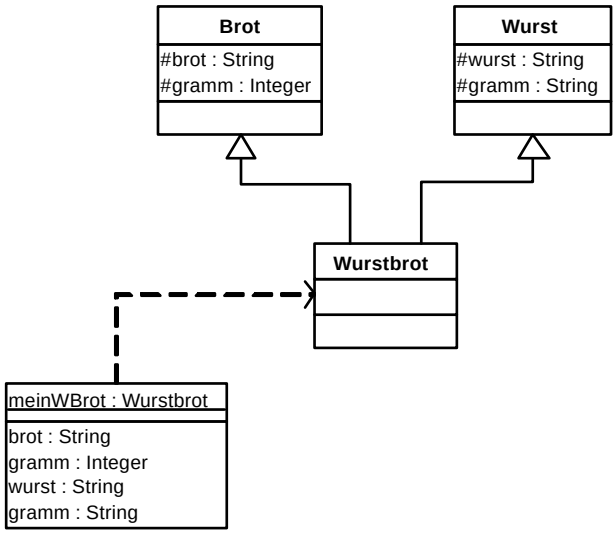

Abbildung 8.20 Mehrfachvererbung - »Wurstbrot« hat zwei Basisklassen.

Sicherlich interessiert es Sie nun, wie das UML-Diagramm aussieht, wenn man hier noch die virtuelle Basisklasse `Supermarkt` hinzufügt, die in Abschnitt 4.9.2, »Virtuelle indirekte Basisklassen erben«, beschrieben wurde. In Abbildung 8.21 finden Sie das entsprechende UML-Diagramm.

[»] **Hinweis**

Es sollte noch erwähnt werden, dass Programmiersprachen wie Java und C# keine Mehrfachvererbung implementiert haben. Berücksichtigen Sie dies, wenn Sie ein Projekt in dieser Sprache planen. Als Alternative zur Mehrfachvererbung kann man – je nach Anwendungsfall – Schnittstellen oder eine Mischung aus Assoziationen und Einfachvererbung verwenden.

Abstrakte Klassen

In Abschnitt 4.8.7, »Rein virtuelle Methoden und abstrakte Basisklassen«, wurden die abstrakten Klassen behandelt. Hier haben Sie auch erfahren, dass man von abstrakten Klassen niemals ein Objekt erzeugen kann. Diese Klassen werden vorwiegend verwendet, um zum Beispiel eine gemeinsame Funktionalität für mehrere (abgeleitete) Klassen bereitzustellen oder wenn man eine Methode auf jeden Fall redefinieren will. Eine abstrakte Klasse zeigen Sie in UML an, indem Sie den Klassennamen *kursiv* schreiben.

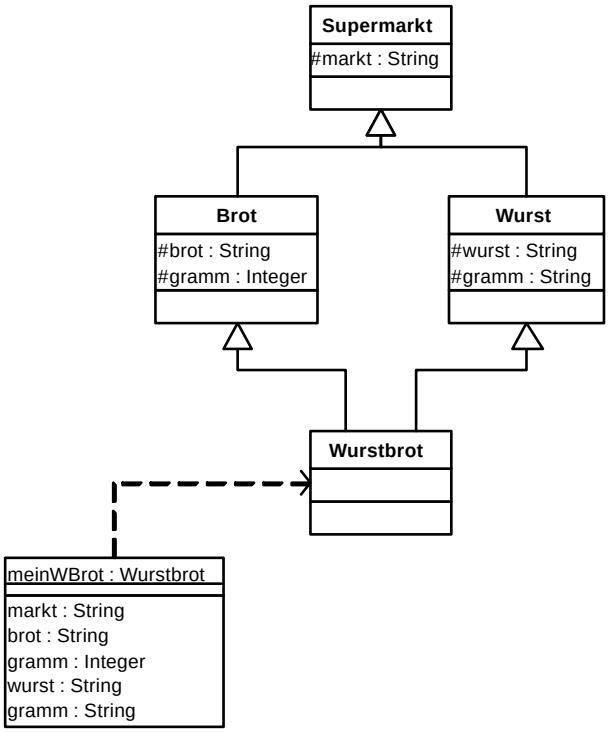

Abbildung 8.21 Die virtuelle indirekte Basisklasse »Supermarkt«

Assoziationen

Eine strukturelle Beziehung zwischen Klassen nennt man eine *Assoziation*. Über diese Assoziation kommunizieren die dadurch verbundenen Objekte miteinander. Assoziationen machen praktisch Objekte miteinander bekannt. Eine solche Verbindung wird auch als *Link* (dt.: Bindeglied) bezeichnet. Eine einfache (binäre) Assoziation kann in der Praxis folgendermaßen gebildet werden:

Abbildung 8.22 UML-Darstellung einer Assoziation

Eine Assoziation wird beim Klassendiagramm mit einer einfachen Verbindungslinie gezeichnet, um die Klassen miteinander zu verbinden. Mit `Assoziation` geben Sie hier den Assoziationsnamen an. Die Leserichtung wird neben diesem Namen mit einer Pfeilspitze eingezeichnet. Jede Klasse in der Assoziation hat eine bestimmte Rolle (hier mit `End1` und `End2`). Die Sterne unterhalb der Verbin-

dungslinie geben die *Kardinalität* der Assoziation an. Damit gibt man an, wie viele Objekte der einen Klasse mit einem Objekt der anderen Klasse in Verbindung stehen.

[»] **Hinweis**

Es sollte noch darauf hingewiesen werden, dass selten alle Angaben einer Assoziation spezifiziert werden. Es ist außerdem auch sehr unübersichtlich, wenn man jede noch so bedeutungslose Assoziation abbildet.

Hierzu ein einfaches Beispiel: Ein `Autor` schreibt ein Buch für einen `Verlag`:

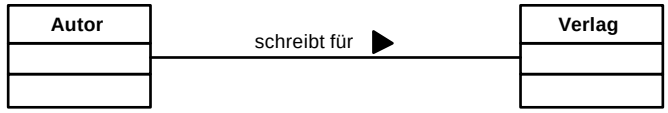

Abbildung 8.23 Assoziation zwischen Autor und Verlag

Die Rollen lassen sich recht einfach verteilen, der Autor ist der Arbeitnehmer und der Verlag der Arbeitgeber. In Abbildung 8.24 können Sie sehen, wie diese Rollen eingetragen werden.

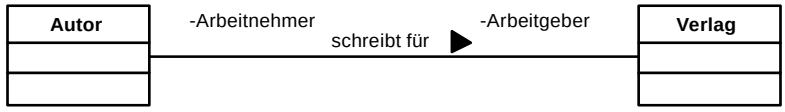

Abbildung 8.24 Jede Klasse hat in der Assoziation auch eine Rolle.

Natürlich kann eine solche Assoziation auch in der umgekehrten oder auch in zwei Richtungen vorhanden sein. Gibt der Verlag einen Auftrag an den Autor, dann ergibt sich folgendes UML-Diagramm:

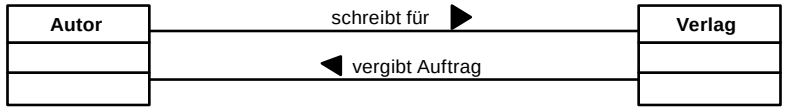

Abbildung 8.25 Zwei Assoziationen zwischen den Klassen

Assoziationen sind aber nicht nur zwischen zwei Klassen möglich. So können Klassen auch mit mehreren Klassen zusammenhängen. Denkt man da wieder an das Beispiel mit dem Verlag, so muss man berücksichtigen, dass hier auch ein Lektor arbeitet, der, abgesehen vom Schreiben des Buches, die ganze restliche Arbeit am Manuskript erledigt. Dann ist auch noch der Drucker beteiligt, der aus dem Dokument ein Buch macht. Daraus ergibt sich Abbildung 8.26.

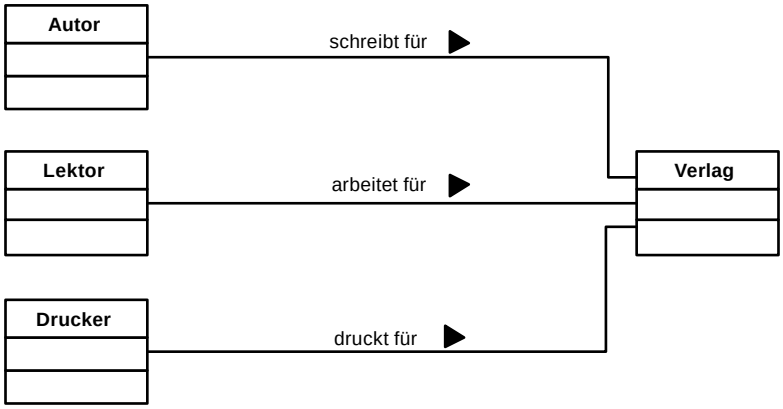

Abbildung 8.26 Hier ist eine Klasse mit mehreren Klassen assoziiert.

Häufig ergibt sich aus einer Assoziation zwischen zwei Klassen eine Regel bzw. Einschränkung. So untersucht ein Doktor zwar viele Patienten, aber niemals alle auf einmal, sondern immer einen nach dem anderen. Eine solche Einschränkung wird in der Nähe der betreffenden Klasse zwischen geschweiften Klammern vermerkt (siehe Abbildung 8.27).

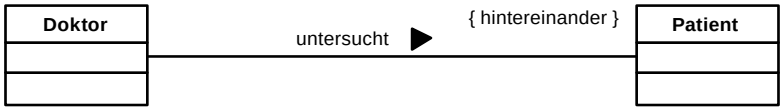

Abbildung 8.27 Eine Assoziation mit Einschränkung

Eine weitere Einschränkung ist die *Oder-Beziehung*. Diese wird durch eine gestrichelte Linie und die Einschränkung in den geschweiften Klammern dargestellt. Die gestrichelte Linie verbindet die beiden Assoziationslinien miteinander. Im nächsten Beispiel wählt ein Gast in einem Lokal als Vorspeise entweder Salat oder Suppe (siehe Abbildung 8.28).

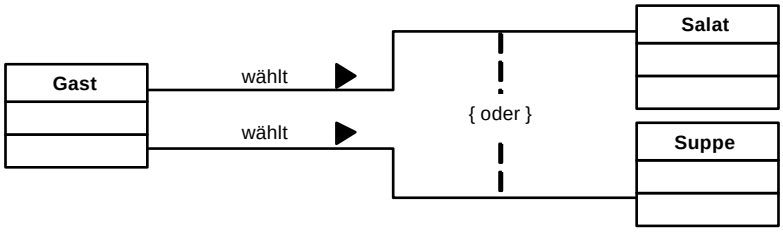

Abbildung 8.28 Eine Oder-Beziehung zwischen den Assoziationen

Da eine Assoziation ebenfalls Attribute und Operationen wie die Klasse haben kann, gibt es für solche Fälle eine Assoziationsklasse, die wie eine normale Klasse visualisiert wird. Außerdem wird dabei die Assoziationslinie mit einer gestrichelten Linie zur Assoziationsklasse verbunden. In Abbildung 8.29 ist `Gehalt` die Assoziationsklasse.

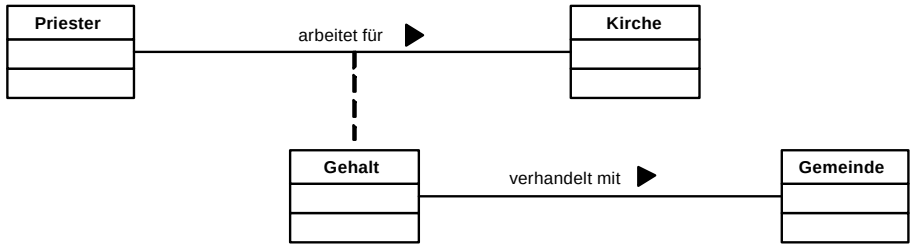

Abbildung 8.29 Eine Assoziationsklasse »Gehalt«

Wie Klassen haben auch die daraus erzeugten Objekte eine Assoziation. In diesem Fall spricht man von einer Verknüpfung (Link). Die Anwendung verläuft wie bei der Assoziation für Klassen, nur dass man die Namen der Objekte und auch die Assoziation unterstreicht.

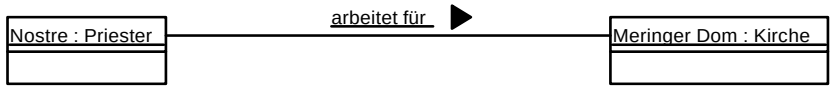

Abbildung 8.30 Eine Verknüpfung zwischen zwei Objekten

Kardinalität

Die bisherigen Beispiele haben den Eindruck vermittelt, dass die Verbindung zwischen den Klassen immer eins zu eins ist. Rein logisch betrachtet ist dies aber nicht immer so. Beispielsweise untersucht ein Doktor, wie erwähnt, nicht nur einen Patienten am Tag, sondern einen bis beliebig viele (und natürlich nicht gleichzeitig). Diese Angaben werden mit einer Kardinalität (oder auch *Multiplizität*) angegeben. Dabei wird die Anzahl neben der entsprechenden Klasse oberhalb der Assoziationslinie gesetzt:

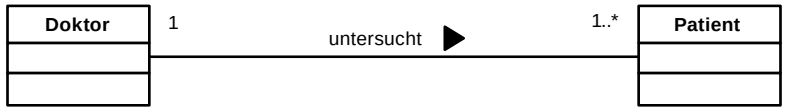

Abbildung 8.31 Kardinalität zwischen zwei Klassen

Dabei gibt es viele unterschiedliche Möglichkeiten, eine solche Kardinalität darzustellen. Wie Sie in Abbildung 8.31 sehen konnten, werden für ein ODER zwei Punkte verwendet. In Worten: Ein Doktor untersucht einen oder beliebig (*) viele Patienten. Das Sternchen wird benutzt, um mehrere oder viele zu kennzeichnen. Beispielsweise spielen mindestens elf oder mehrere Fußballer für eine Mannschaft. Dies wird folgendermaßen abgebildet (siehe Abbildung 8.32):

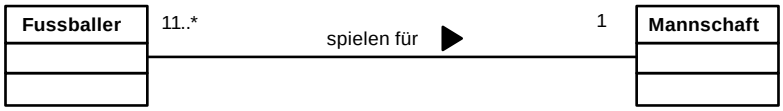

Abbildung 8.32 Kardinalität zwischen Fußballer(n) und einer Mannschaft

Ein neues Beispiel: Eine Person fährt mit einem Auto. Natürlich ist es auch möglich, dass eine Person nicht mit einem Auto fährt. Auch möglich ist zudem, dass mehrere Personen in einem Auto mitfahren. Das folgende Beispiel (siehe Abbildung 8.33) zeigt, dass eine oder maximal vier Personen mit einem oder keinem (0) Auto fahren können.

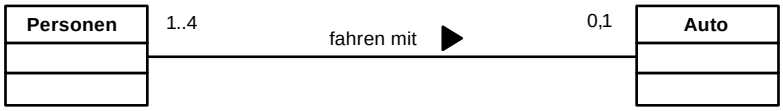

Abbildung 8.33 Kardinalität zwischen einer Person und einem Auto

Natürlich lässt sich dies beliebig weiterspinnen. In Tabelle 8.5 finden Sie einige Möglichkeiten, die Sie für die Kardinalität verwenden können.

Kardinalität	Beschreibung
1	eine Verbindung mit einem Exemplar einer Klasse
0..1	eine Verbindung mit keinem oder einem Exemplar einer Klasse
*	eine Verbindung mit einem, keinem oder vielen Exemplaren einer Klasse
3..*	eine Verbindung mit drei bis vielen Exemplaren einer Klasse
7	eine Verbindung mit genau sieben Exemplaren einer Klasse
2..4, 8..10, 14..*	eine Verbindung mit zwei bis vier, acht bis zehn oder 14 bis beliebig vielen Exemplaren einer Klasse
3, 6, 9	eine Verbindung mit drei, sechs oder neun Exemplaren einer Klasse

Tabelle 8.5 Kardinalitäten

Spezielle Assoziationen

Es ist auch möglich, dass eine Klasse von sich selbst assoziiert ist – auch *reflexive Assoziation* genannt. Beispielsweise können Objekte einer Klasse `Personal` sowohl die Rolle eines Vorgesetzten als auch von Mitarbeitern übernehmen. So kann zum Beispiel ein oder kein Vorgesetzter einen oder beliebig viele Mitarbeiter führen. Eine solche reflexive Assoziation kann wie folgt abgebildet werden (siehe Abbildung 8.34):

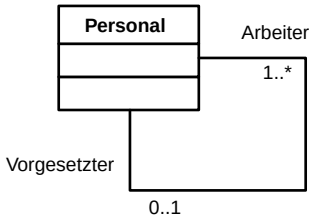

Abbildung 8.34 Reflexive Assoziation

Manchmal benötigt man so etwas wie eine eindeutige Identifikation, ein Attribut, das dazu verwendet werden soll, ein Objekt in eine Gruppe zu unterteilen. Dazu kann man eine *qualifizierte Assoziation* verwenden. Eine solche Assoziation stellt die Verbindung zwischen den Objekten der Klassen über ein qualifiziertes Attribut her. Wenn Sie eine Klasse `Personal` und eine Klasse `Computer` haben, so kann das Personal auf den Computer zugreifen. Aber da es nun mal viel Personal gibt, soll nicht jeder einfach auf den Computer zugreifen können. Daher benötigt das Personal ein Passwort. Das Passwort stellt hierbei die qualifizierte Assoziation zwischen den Klassen her. Diese Assoziation wird in einem kleinen Rechteck an der betreffenden Klasse angebracht (siehe Abbildung 8.35).

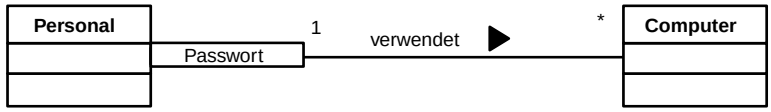

Abbildung 8.35 Qualifizierte Assoziation

In diesem Beispiel haben Sie außerdem die »eins-zu-viele«-Kardinalität in eine »eins-zu-eins«-Kardinalität umgemünzt.

Ein Mitarbeiter arbeitet zum Beispiel in einer Abteilung. In der Regel haben die Mitarbeiter eine bestimmte Tätigkeit (Funktion) in ihrem jeweiligen Bereich, für den sie verantwortlich sind. Durch diese Tätigkeit kann man die einzelnen Mitarbeiter einer Abteilung in verschiedene Gruppen einsortieren. Grafisch kann man diese qualifizierte Assoziation folgendermaßen abbilden:

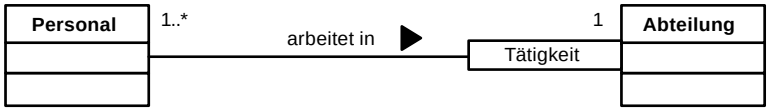

Abbildung 8.36 Qualifizierte Assoziation

Hinweis [«]

Natürlich ließe sich noch weitaus mehr zu den Assoziationen schreiben, aber für den Normalgebrauch sind Sie hiermit gerüstet.

Aggregationen

Eine weitere Form einer Beziehung ist die *Aggregation*, die dann vorliegt, wenn sich ein Objekt aus einem oder mehreren Objekten zusammensetzt. Man spricht dabei von einer *Teil-Ganzes-Beziehung*. Hierbei wird aber auch eine Rangordnung angegeben. Die höheren Objekte setzen sich aus Objekten niedrigeren Rangs zusammen, bzw. Objekte von niedrigem Rang sind ein Teil eines Objekts von höherem Rang.

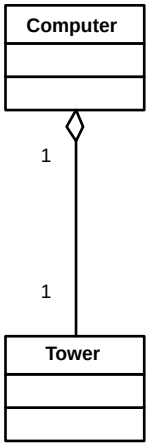

Abbildung 8.37 Eine einfache Aggregation

In Abbildung 8.37 sehen Sie eine einfache Aggregation, die zwei Klassen miteinander verbindet. Die nicht ganz ausgefüllte Raute steht immer an der Klasse, die ein Ganzes darstellt und sich aus den Teilen zusammensetzt. Jeder, der sich ein wenig mit dem Innenleben eines Computers auskennt, weiß, dass der Tower wiederum nur Teil eines Ganzen ist. Im Tower befindet sich zum Beispiel eine Festplatte, ein CD/DVD-Laufwerk, ein Mainboard und so weiter. Wobei das Mainboard wiederum in einzelne Teile zergliedert werden kann. So befindet sich

auf dem Mainboard beispielsweise der Prozessor oder der Arbeitsspeicher. Abbildung 8.38 zeigt eine solche hierarchische Aggregation eines Computers.

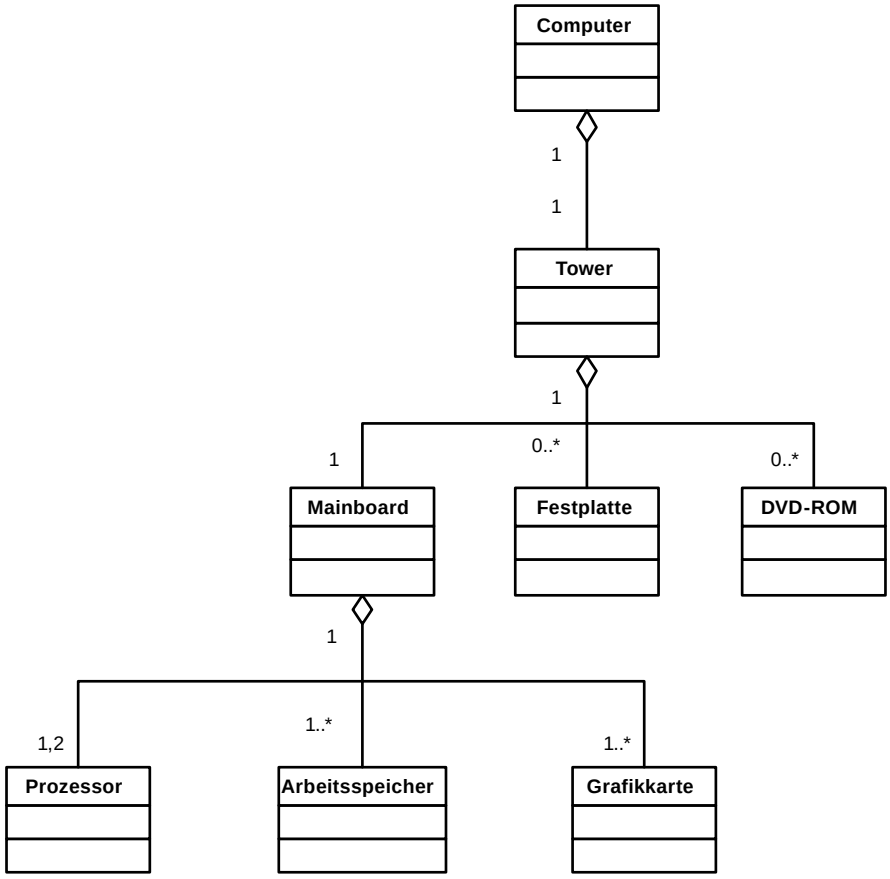

Abbildung 8.38 Eine mehrstufige Aggregation

In Abbildung 8.38 wird zudem der wichtige Zusammenhang von Kardinalität und Aggregation deutlich. In einem Tower befindet sich zum Beispiel ein Mainboard. Auf diesem Mainboard (abhängig vom Typ) können beispielsweise ein oder maximal zwei Prozessoren vorhanden sein. Abhängig davon, wie viele RAM-Bänke auf dem Mainboard sind, können in ein Mainboard ein oder mehrere RAM-Riegel gesteckt werden. Dasselbe Bild ergibt sich mit der Grafikkarte. Hierbei lassen sich aber auch gleich Einschränkungen einbauen. Wollen Sie die Option angeben, dass in den Computer entweder ein DVD-ROM- oder ein DVD-RAM-Laufwerk kommt, können Sie, wie gewohnt, eine Oder-Verknüpfung einbauen (siehe Abbildung 8.39).

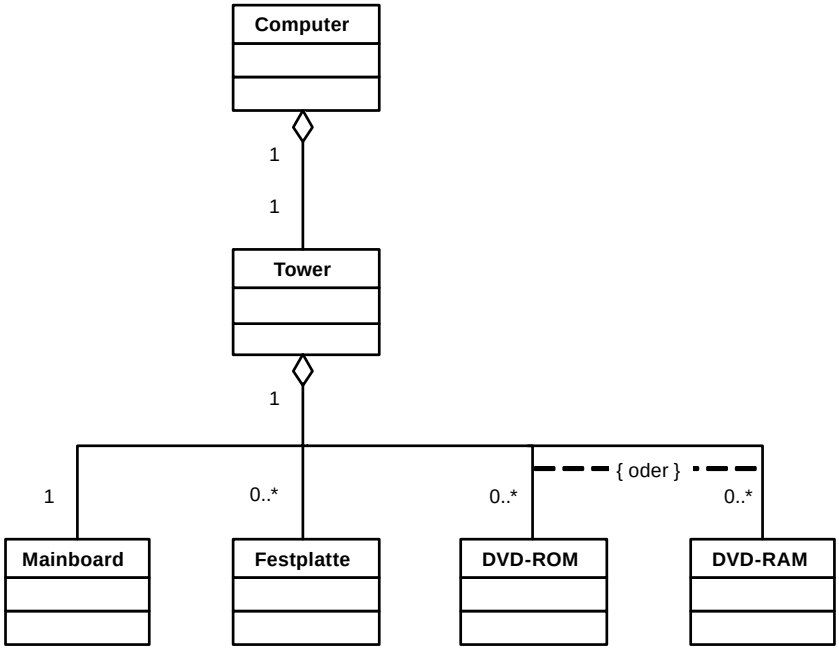

Abbildung 8.39 Eine Aggregation mit Einschränkung

Komposita – eine strenge Aggregation

Ein Kompositum ist im Grunde (und in der objektorientierten Welt sowieso) auch nur eine Form der (strengen) Aggregation. Abbildung 8.40 soll den Unterschied verdeutlichen.

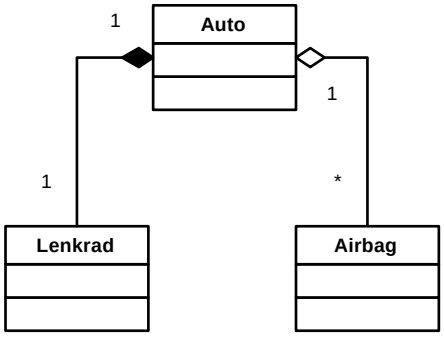

Abbildung 8.40 Ein Kompositum und eine Aggregation

Ein Kompositum wird ebenfalls mit einer Raute dargestellt, nur diesmal mit einer ausgefüllten Raute. Das Beispiel enthält die Aussage, dass ein Auto zwingend ein Lenkrad benötigt. Aber ein Auto muss nicht zwingend einen Airbag haben, das

heißt, ein Auto kann auch ohne einen Airbag gesteuert werden (bzw. existieren). Wenn also aus dem Auto das Lenkrad ausgebaut wird, kann es nicht mehr gefahren werden. Fehlt hingegen der Airbag, ist dies kein Problem, und das Auto kann trotzdem gesteuert werden. Also *muss* das Lenkrad Teil eines Ganzen (hier des Autos) sein, und der Airbag *kann* Teil eines Ganzen sein.

Kontexte

Wenn Sie mehrere Gruppen von Klassen als Aggregation oder Komposita modellieren müssen, kann es oft recht unübersichtlich werden. Will man dann die Aufmerksamkeit auf eine Gruppe lenken, kann hierzu ein Kontextdiagramm zur Modellierung verwendet werden. Ein solches Kontextdiagramm kann man sich wie den Ausschnitt aus einer Landkarte vorstellen. Besonders im Zusammenhang mit Komposita spielen solche Kontextdiagramme eine wichtige Rolle. Beispielsweise können wir bei einem Computer auf diese Weise den Computer zunächst als Ganzes beschreiben und trotzdem auf seine inneren Bestandteile eingehen (siehe Abbildung 8.41).

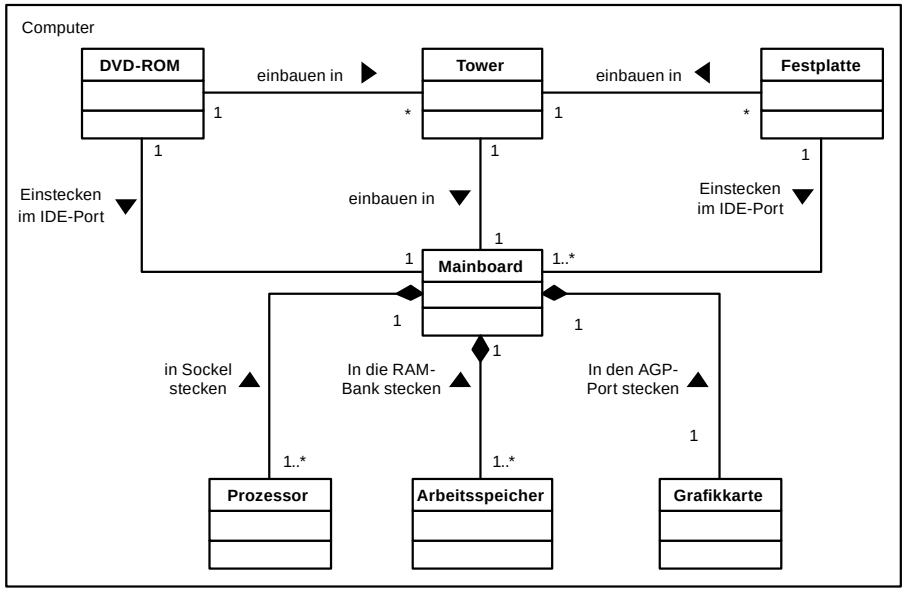

Abbildung 8.41 Ein Kontextdiagramm fasst kleine Klassen zusammen.

Wenn Sie eine solche Detailaufnahme gemacht haben, können Sie damit auch wieder andere Klassen verbinden und so einen viel besseren Überblick erhalten (siehe Abbildung 8.42).

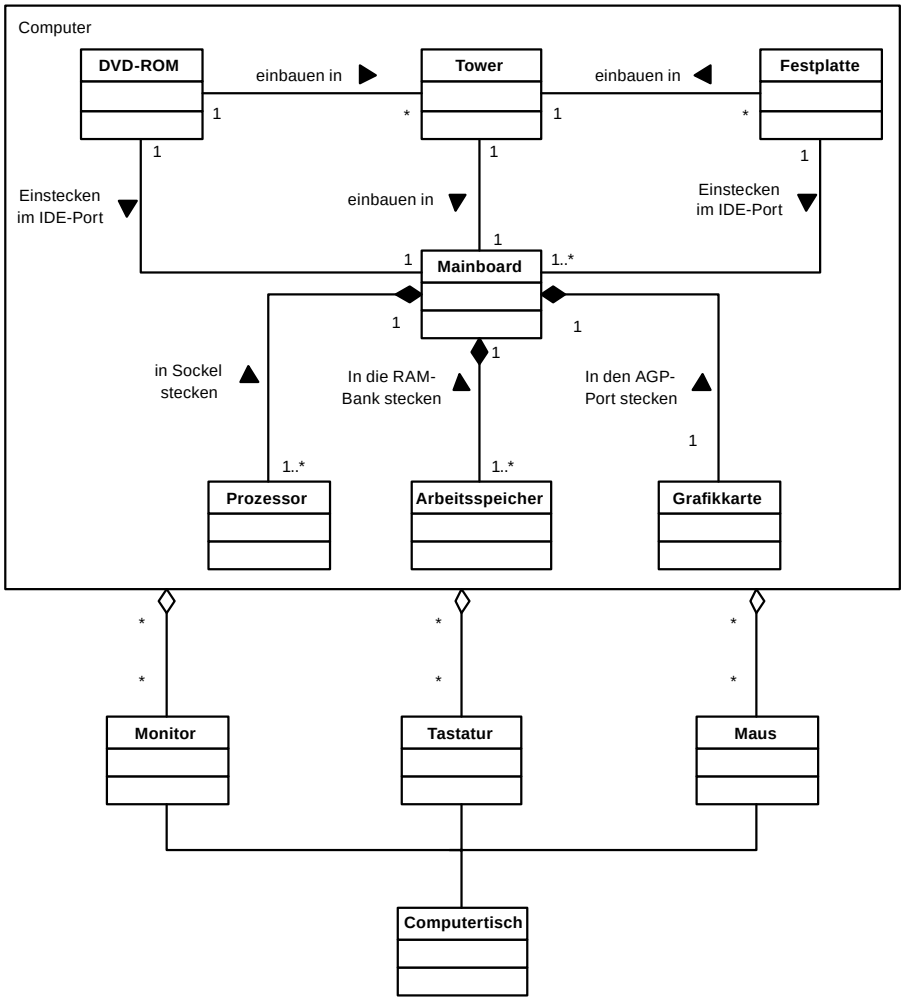

Abbildung 8.42 Ein Kontextdiagramm zeigt die Bestandteile einer Klasse und wie diese Klasse mit anderen Klassen zusammenhängt.

Pakete

Im Beispiel zuvor konnten Sie bereits erkennen, dass es bei einer ganzen Menge von Klassen schnell unübersichtlich werden kann. Hierbei können in einem Projekt schon einmal an die 100 Klassen zusammenkommen. Damit man den Überblick behält, verwendet man Pakete. So stellt jedes Paket ein Teilmodell dar. Natürlich setzt dies voraus, dass solche Pakete sinnvoll gegliedert sind und nicht wahllos zusammengeworfen werden. Nimmt man zum Beispiel den Computer aus dem letzten Abschnitt und will diesen in einen Netzwerkplan mit einbauen, dann sind die inneren Teile weniger von Interesse. So kann man einfach alle Klas-

sen innerhalb des Computers zu einem Paket zusammenfassen. Bezogen auf die Abbildung 8.42, sieht dies wie in Abbildung 8.43 dargestellt aus.

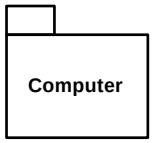

Abbildung 8.43 Alle Klassen eines Computers zu einem Paket zusammengefasst

Gewöhnlich greifen solche Klassen eines Pakets auch auf andere Klassen zu, die wiederum als Paket dargestellt werden können. Dies geschieht über Vererbungsbeziehungen und Assoziationen. Natürlich setzt dies voraus, dass die einzelnen Klassen, die zum Paket `Computer` gehören, dann auch entsprechend kenntlich gemacht werden, um diese identifizieren zu können. Dies wird erledigt, indem man dem Klassennamen den Paketnamen voranstellt (siehe Abbildung 8.44).

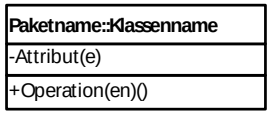

Abbildung 8.44 Paketzugehörigkeit in UML

Natürlich ist es auch möglich und gängige Praxis, dass zwischen den Paketen eine Vererbungs- und Abhängigkeitsbeziehung besteht (siehe Abbildung 8.45):

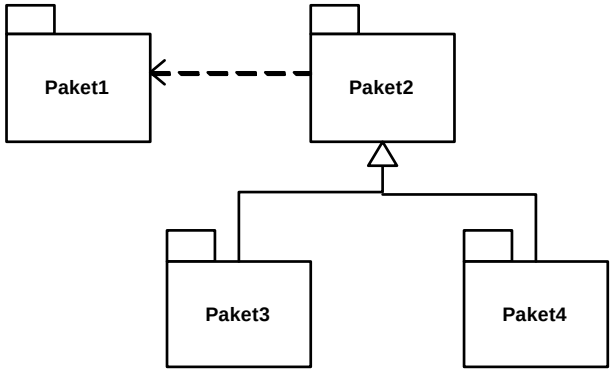

Abbildung 8.45 Abhängigkeitsbeziehungen zwischen Paketen

In diesem Beispiel erbt eine Klasse, die zu `Paket3` gehört, von einer Klasse aus `Paket2`. Dasselbe gilt bei `Paket4`. Außerdem ist `Paket2` abhängig von `Paket1`, das heißt, `Paket2` würde ohne `Paket1` nicht voll funktionieren. Auf die Abhängigkeiten, die mit der gestrichelten Linie gekennzeichnet werden, wird im Folgenden

noch eingegangen. Durch eine solche Paketansicht wird von vornherein recht übersichtlich dargestellt, welche Vererbungs- bzw. Abhängigkeitsbeziehungen zwischen den Klassen und Schnittstellen vorliegen.

Schnittstellen

Wenn Sie häufig und viele Klassen entwerfen und modellieren, werden Sie das ein oder andere Mal eine Klasse erstellen, die zwar im Grunde nichts mit einer Oberklasse gemeinsam hat, aber dennoch recht ähnliche oder zum Teil dieselben Operationen mit derselben Signatur verwendet. In der Praxis werden Sie diese Operationen in einer Klasse erstellen, die von anderen Klassen verwendet werden kann. Sie haben also eine Klasse mit vielen wiederverwendbaren Operationen, die in UML über eine Schnittstelle (*interface*) modelliert werden. Eine Schnittstelle ist im Grunde nichts anderes als eine Liste von Attributen und Operationen, die eine öffentliche Sichtbarkeit haben.

Das Implementieren einer Schnittstelle bedeutet, dass die Klasse verpflichtet ist, diese Operation(en) zu übernehmen, und komplett die Definition der Aufgabenlösung beschreibt.

In UML stehen Ihnen für Schnittstellen zwei verschiedene Notationsformen zur Verfügung, die Lollipop-Darstellung und die klassenähnliche Notation. Bei der Lollipop-Darstellung wird die Schnittstelle als nicht ausgefüllter Kreis dargestellt, mit dem die Klassen, die die Schnittstelle implementieren, über eine Linie verbunden werden (siehe Abbildung 8.47).

Die zweite Möglichkeit der Darstellung ist die klassenähnliche, bei der die Schnittstellen mit dem Stereotyp »interface« gekennzeichnet werden. Eine solche Klasse enthält keinerlei Attribute. Bei dieser Form der Darstellung ist auch die Aufrufkonvention der Operation(en) erkennbar (siehe Abbildung 8.46).

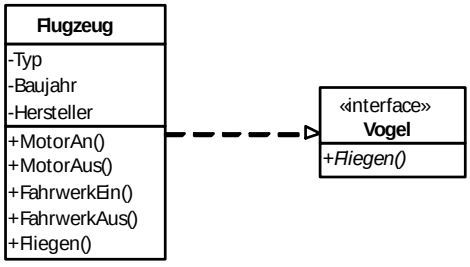

Abbildung 8.46 Darstellung einer klassenähnlichen Schnittstelle

Hier stellt die Schnittstelle `Vogel` per Aufrufkonvention eine Operation `Fliegen()` zur Verfügung, und die Klasse `Flugzeug` implementiert diese Operation.

Dasselbe wird in der Lollipop-Darstellung folgendermaßen präsentiert:

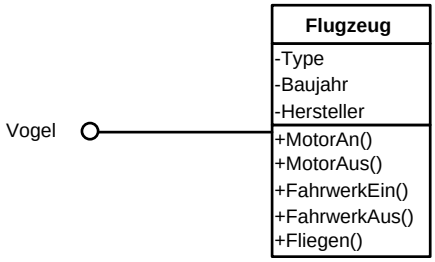

Abbildung 8.47 Eine verkürzte Darstellung einer Schnittstelle

Wenn jetzt andere Klassen auf diese Schnittstelle zugreifen, wird diese Abhängigkeit durch eine gestrichelte Linie mit einem Pfeil gezeichnet (siehe Abbildungen 8.48 und 8.49).

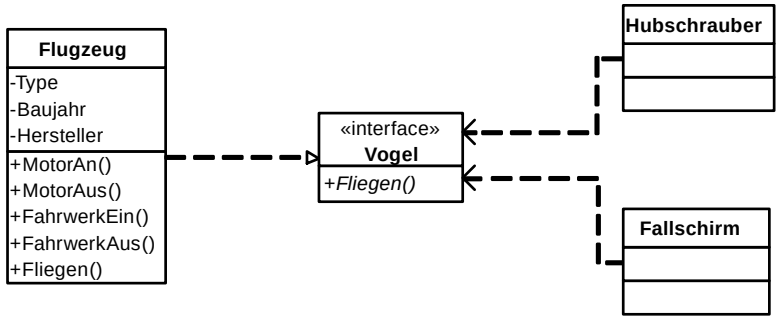

Abbildung 8.48 Verwendung einer Schnittstelle (Möglichkeit 1)

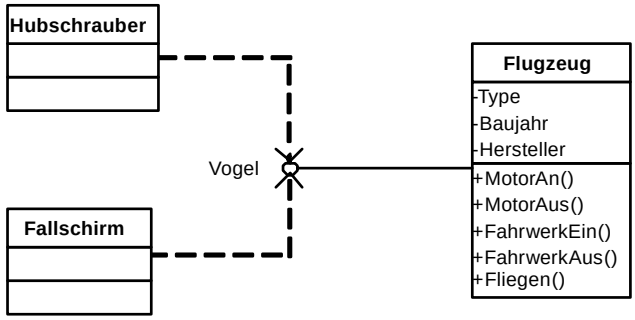

Abbildung 8.49 Verwendung einer Schnittstelle (Möglichkeit 2)

Beim Betrachten dieser beiden Abbildungen finde ich, dass man an der klassenähnlichen Darstellung mehr erkennen kann. Vor allem wird hier gleich sichtbar,

welche Schnittstelle(n) implementiert ist (sind). Es spricht auch nichts dagegen, aus einer Klasse mehrere Schnittstellen bereitzustellen (siehe Abbildung 8.50).

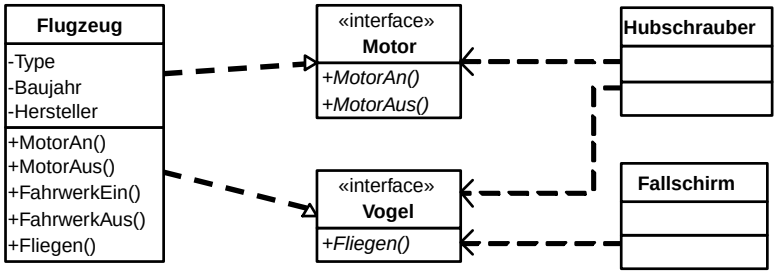

Abbildung 8.50 Mehrere Schnittstellen bereitgestellt

Hier vereinbart die Schnittstelle `Motor` durch die Aufrufkonvention die Operationen `MotorAn` und `MotorAus`. Die zweite Schnittstelle legt die Aufrufkonvention für die Operation `Fliegen` fest. Die Klasse `Flugzeug` implementiert hingegen diese Operationen und definiert alle drei vollständig.

Abhängigkeiten

Im letzten Abschnitt, in dem die Schnittstellen behandelt wurden, wurde auch die Abhängigkeit besprochen und angewendet. Von einer Abhängigkeit spricht man, wenn sich eine Änderung eines unabhängigen Elements auf das abhängige Element auswirken kann, aber nicht muss. Oder einfacher, wenn der Programmierer Änderungen einbringt und es wurde eine Abhängigkeit modelliert, dann müssen die Elemente auf deren Kompatibilität hin geprüft werden. Daher kann man fast immer davon ausgehen, dass ein Klasse eine andere Klasse benutzt, wenn eine Abhängigkeit modelliert wird.

In den Abbildungen des letzten Abschnitts (Schnittstellen) konnten Sie bereits sehen, dass solche Abhängigkeiten durch einen gestrichelten Pfeil gekennzeichnet werden. Man verbindet dabei immer ein abhängiges Element mit einem unabhängigen Element.

Abbildung 8.51 Eine gestrichelte Linie mit Pfeil kennzeichnet die Abhängigkeit.

Abhängigkeitsbeziehungen können bestehen zwischen

- zwei Klassen
- zwei Paketen
- einer Operation und einer Klasse
- einer Klasse und einer Schnittstelle

Die Praxis

Auf diese Weise können Sie nicht nur »tolle Bildchen« malen und andere damit vielleicht beeindrucken – viel wichtiger dürfte für Sie wohl sein, wie man dies in der Praxis einsetzt. Wie wird aus einem Gespräch ein UML-Modell und am Ende ein komplettes Programm? Wie man komplette Programme schreibt, wissen Sie ja durch dieses Buch mittlerweile. Aber wie aus einem Gespräch ein UML-Entwurf entsteht, soll jetzt ein wenig erläutert werden.

Gespräch analysieren

Zuerst steht immer das Gespräch mit dem Kunden oder den Kollegen an. In diesem Fall soll nochmals die Autovermietung Thema sein. Beim ersten Gespräch ist es zunächst wichtig, dass Sie im Kopf (oder auf einem Blatt Papier) festhalten, was später Klassen und was Attribute werden, welche Operationen verwendet werden und wie die Beschriftung der Assoziation aussieht. Zugegeben, es ist kein leichtes Unterfangen, aber es geht.

Ein kurzer Ausschnitt als Beispiel: Der Kunde geht an die Rezeption der Autovermietung und wird dort gebeten zu warten, bis ein Bearbeiter Zeit für ihn hat. Das Personal an der Rezeption gibt einem Bearbeiter Bescheid, dass ein Kunde auf ihn wartet, um ein Auto zu mieten.

Aus diesem kleinen Ausschnitt würden sich folgende Klassen bilden lassen:

[Kunde], [Rezeption], [Bearbeiter], [Auto]

Als Assoziation (bzw. künftige Operationen) kann man verwenden:

(Kunde) wird empfangen an (Rezeption), (Kunde) wartet auf (Bearbeiter), (Rezeption) informiert (Bearbeiter), (Kunde) will mieten (Auto)

Sie sehen, aus einem einfachen Gespräch lassen sich bereits die ersten Elemente zusammensetzen.

Klassendiagramm entwickeln

Die erste und wichtigste Aufgabe eines UML-Modellierers beginnt mit der Erstellung eines Klassendiagramms. Am besten würfelt man hier zunächst einfach alles zusammen, woraus Sie eine Klasse machen würden. Dabei kann es durchaus vorkommen, dass Klassen hinzugefügt bzw. entfernt werden. Während dieser Phase werden Sie sich immer stärker mit dem Projekt befassen und vor allem ein immer tieferes Verständnis dafür entwickeln. Hierbei werden Sie das ein oder andere Mal nachfragen müssen. So ist es zum Beispiel wichtig zu wissen, ob die Firma einen eigenen Mechaniker beschäftigt oder nicht. Wenn nicht, muss man sich keine Gedanken um die Klasse `Mechaniker` machen. Oder ist im Gespräch

von der Bezahlung die Rede, so ergibt sich hierbei vielleicht die Möglichkeit, mit Kreditkarte oder bar (oder noch auf andere Art) zu zahlen. Auch hier tun Sie gut daran zu hinterfragen, welche Modalitäten möglich sind.

Nach dem Abschluss des Gesprächs sollten Sie beginnen, in einer Liste festzuhalten, was sich alles für eine Klassenbildung eignen würde. Versuchen Sie noch nicht, die einzelnen Klassen zu gruppieren. Ein erster Entwurf eines Klassendiagramms könnte demnach so aussehen (siehe Abbildung 8.52):

Abbildung 8.52 Der erste Klassenentwurf für das Diagramm

Zu einer Autovermietung gehören noch weitere Klassen, aber es soll eben nur ein einfaches Beispiel sein.

Klassen gruppieren

Als Nächstes ist es sehr wichtig, aus diesen Klassen sinnvolle Gruppen zu bilden. Hierbei gibt es zwar keine Regeln, aber von jetzt an ist eine gute Gruppierung entscheidend für den Programmentwurf.

Hierbei könnten man die Klassen `Rezeption`, `Bearbeiter`, `Bereitsteller` und `Mechaniker` in eine eigene Gruppe stellen.

Eine weitere Gruppe könnte man mit dem Mietgegenstand `Auto` bilden – wobei man hier besser beraten ist, einen Oberbegriff wie `Mietgegenstand` zu verwenden, denn so können wir später ohne großen Aufwand das Programm erweitern, wenn die Firma vielleicht auch LKWs, Anhänger oder Sonstiges vermieten sollte. In der Gruppe `Mietgegenstand` wäre es außerdem sinnvoll, den `Zustand` und die `Ausstattung` als Unterklasse mitzuverwenden. Hier wurde aus der Klasse `Auto` einfach eine abstrakte Klasse gebildet, da nicht die Funktionen des Autos wie `MotorAn()` etc. im Vordergrund stehen. Solche Operationen machen hier (bei einer Autovermietung) wenig Sinn. Wenn man das Ganze noch etwas näher betrachtet, haben selbst Klassen wie `Zustand` und `Ausstattung` wenig Sinn. Da bei einer Vermietung eigentlich nicht die Operationen eines Autos, sondern die Attribute von Bedeutung sind, würde es sich anbieten, alles in eine Klasse `Auto` zu stecken.

Als weitere Gruppe könnte man die komplette `Vermietung` bilden. Allerdings erscheint es sinnvoller, aus `Vermietung` eine abstrakte Klasse zu machen. In dieser Gruppe ließen sich Klassen wie `Versicherung`, `Bezahlung`, `Bargeld`, `Kreditkarte`, `DauerTage`, `Kilometer`, `Rechnung` und `Wechselgeld` zusammenfassen. Wobei sich die Klassen `DauerTage`, `Kilometer` und `Versicherung` eher in eine Oberklasse `Leistungsumfang` einfügen ließen.

Übrig bleiben noch die Klassen `Kunde` und `Führerschein`, die auch in eine eigene Gruppe kommen sollten.

Somit steht der nächste Entwurf fest, wie die Klassen gruppiert werden. Eine mögliche Gruppierung zeigen die Abbildungen 8.53 bis 8.56.

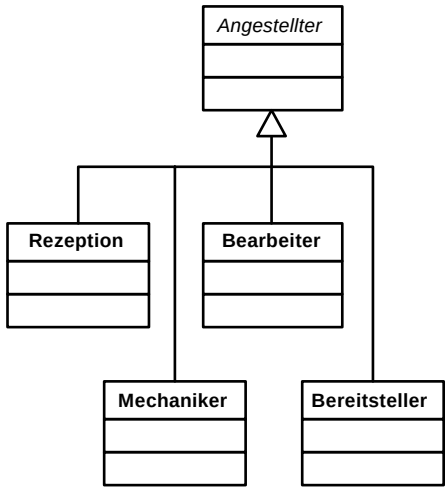

Abbildung 8.53 Eine abstrakte Klasse »Angestellter«

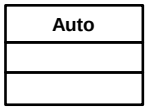

Abbildung 8.54 Die Klasse »Auto« (umfasst jetzt auch den Zustand und die Ausstattung)

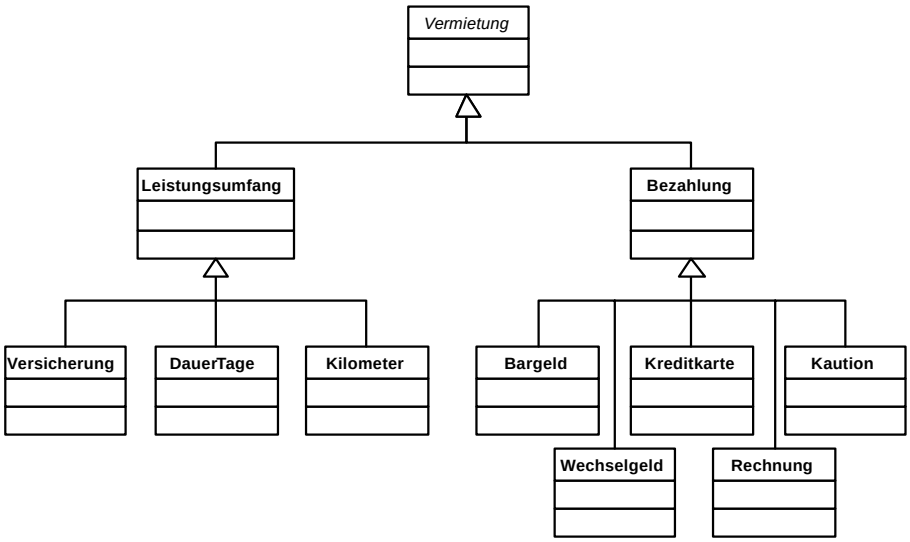

Abbildung 8.55 Abstrakte Klasse »Vermietung«

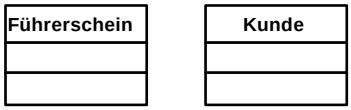

Abbildung 8.56 Die restlichen Klassen

Assoziation bilden

Als Nächstes sollten Sie eine Assoziation zwischen einigen der Klassen bilden. Hierbei ist es wichtig, sich erst einmal auf das Nötigste zu konzentrieren. Verwenden Sie zunächst eine Klasse und überlegen Sie, wie diese mit den anderen Klassen zusammenhängt. Anschließend können Sie dasselbe mit einer anderen Gruppe machen – so lange, bis Sie alle Klassen abgearbeitet haben.

Zunächst soll die Klasse `Kunde` assoziiert werden. Überlegen Sie dazu, womit die Klasse `Kunde` alles assoziiert werden kann. Hierbei wird auch gleich wieder die Frage aufkommen, ob man dies oder jenes wirklich als Klasse implementieren

soll oder überhaupt benötigt. Abbildung 8.57 skizziert einen ersten Überblick dazu.

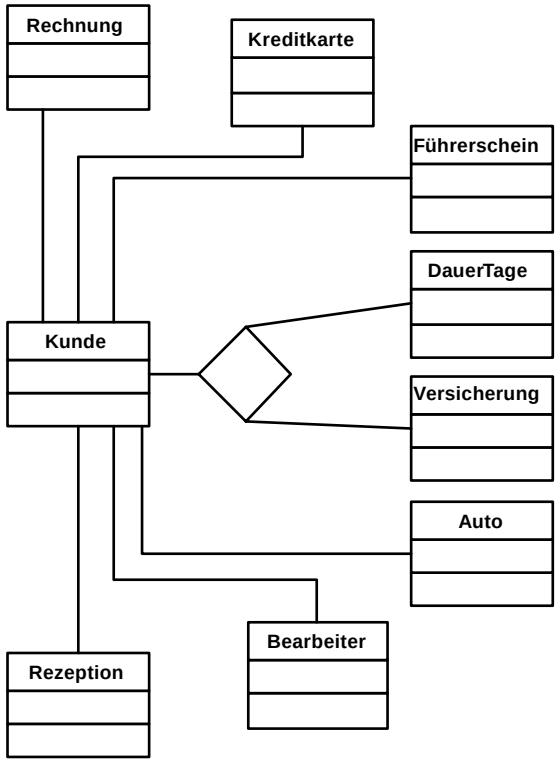

Abbildung 8.57 Eine Assoziation mit der Klasse »Kunde«

Im nächsten Schritt sollten Sie diese Assoziationen beschriften. Betrachtet man Abbildung 8.57, lassen sich folgende Punkte auflisten:

- Der Kunde meldet sich an der Rezeption.
- Der Kunde spricht mit dem Bearbeiter über das Mietobjekt.
- Der Kunde mietet ein Auto.
- Im Leistungsumfang schließt der Kunde eine Versicherung ab und mietet das Auto nach Tagen (`DauerTage`).
- Der Kunde muss noch den Führerschein vorlegen.
- Der Kunde bezahlt mit der Kreditkarte.
- Der Kunde erhält die Rechnung.

Schreibt man dies in das UML-Diagramm, ergibt sich folgendes Bild (siehe Abbildung 8.58):

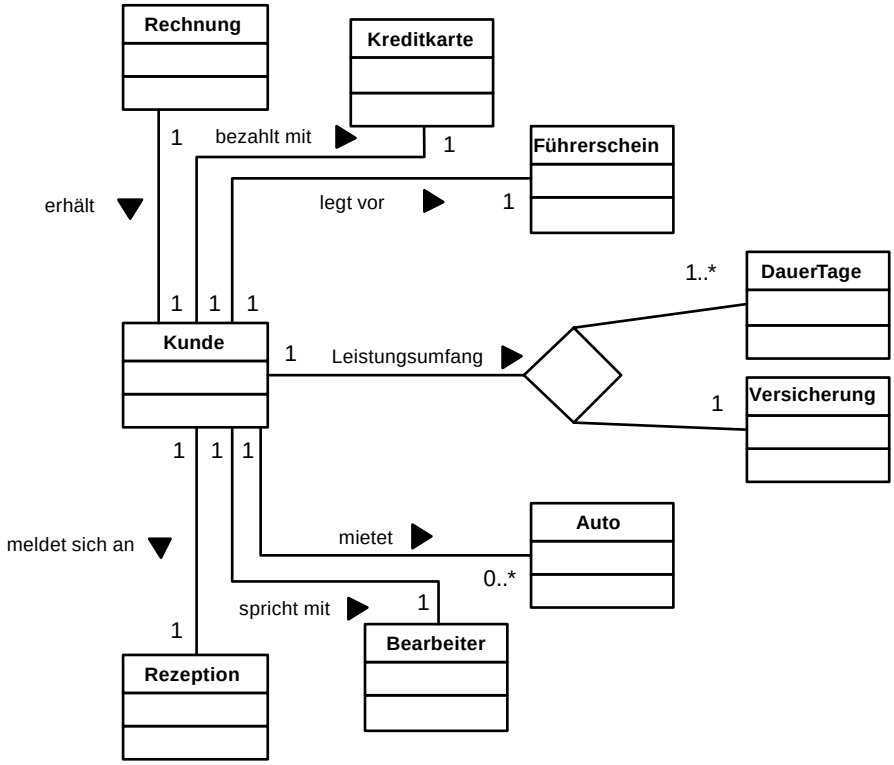

Abbildung 8.58 Eine beschriftete Assoziation mit den Kardinalitäten

Einfacher gestaltet sich die Assoziation der `Rezeption`, die hier nur den Kunden empfangen muss und den Bearbeiter informiert (siehe Abbildung 8.59).

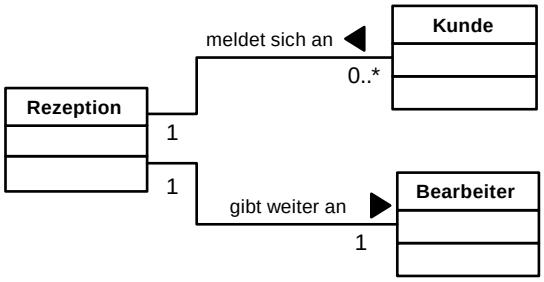

Abbildung 8.59 Beschriftete Assoziation der Rezeption mit den Kardinalitäten

Anders verhält es sich hingegen mit dem Bearbeiter. Seine Aufgaben sind (auf den ersten Blick) diese:

- Bearbeiter wird von Rezeption benachrichtigt (dass der Kunde da ist).
- Bearbeiter empfängt (und begrüßt) Kunden.
- Bearbeiter bietet ein Auto zum Mieten an (und berät den Kunden).
- Bearbeiter bietet Versicherung an.
- Bearbeiter bietet Bezahlung entweder nach Kilometern oder Tagen an.
- Bearbeiter erhält die Bezahlung (vom Kunden).
- Bearbeiter stellt eine Rechnung aus.
- Bearbeiter benachrichtigt Bereitsteller, dass dieser ein Auto zur Verfügung stellt.

Hierzu also die Assoziationen des Bearbeiters:

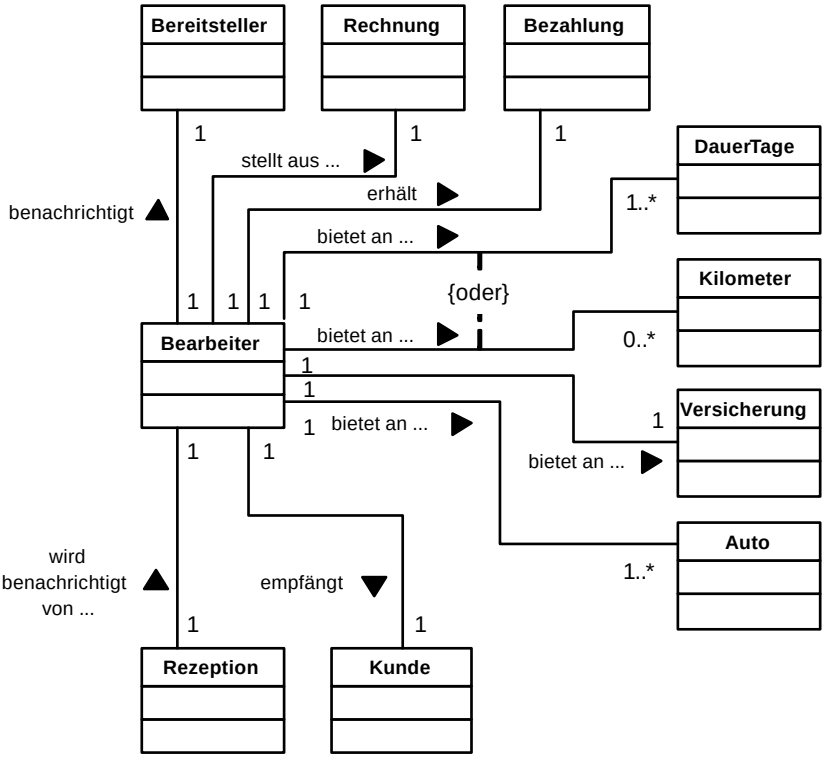

Abbildung 8.60 Beschriftete Assoziation des Bearbeiters mit den Kardinalitäten

Erheblich weniger Assoziationen hat der Bereitsteller des Autos:

- Bereitsteller wird vom Bearbeiter informiert (ein Auto bereitzustellen).
- Bereitsteller übergibt Auto an Kunden.
- Bereitsteller empfängt Auto vom Kunden.
- Bereitsteller stellt (eventuell) Mängel fest.
- Bereitsteller meldet Mängel (oder Auto) beim Bearbeiter.

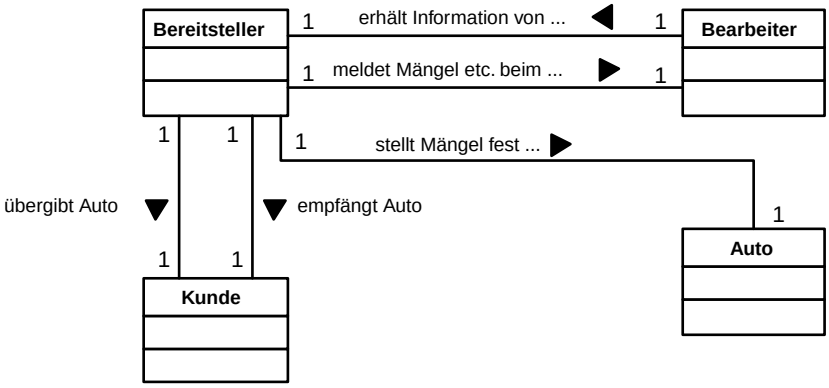

Abbildung 8.61 Beschriftete Assoziation des Bereitstellers mit den Kardinalitäten

Bleibt eigentlich nur noch der Mechaniker, der vom Bearbeiter die Information erhält und die Mängel am Auto behebt. Anschließend meldet der Mechaniker dem Bearbeiter, dass die Mängel am Auto behoben wurden. Hierzu die Assoziationen:

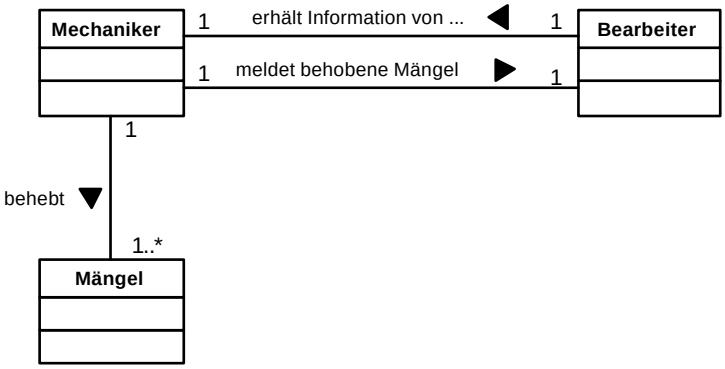

Abbildung 8.62 Beschriftete Assoziation des Mechanikers mit den Kardinalitäten

Aggregationen und Komposita

Nachdem Sie abstrakte Klassen und Assoziationen gebildet haben, sollten Sie nun herausfinden, welche Klassen Bestandteil anderer Klassen sind (Teil eines Ganzen). In unserem Fall ist dies nicht sehr kompliziert. Hierzu könnte man Folgendes vorschlagen:

- Der Leistungsumfang besteht mindestens aus einer Versicherung und der Bezahlung entweder nach Kilometern oder nach Tagen. Kein Auto wird ohne eine Versicherung vermietet.
- Die Bezahlung erfolgt entweder bar oder per Kreditkarte. Bei Barzahlung fällt außerdem eine Kaution an, und gegebenenfalls wird auch Wechselgeld benötigt.

Die hier beschriebenen Aggregationen und Komposita können Sie wie folgt modellieren:

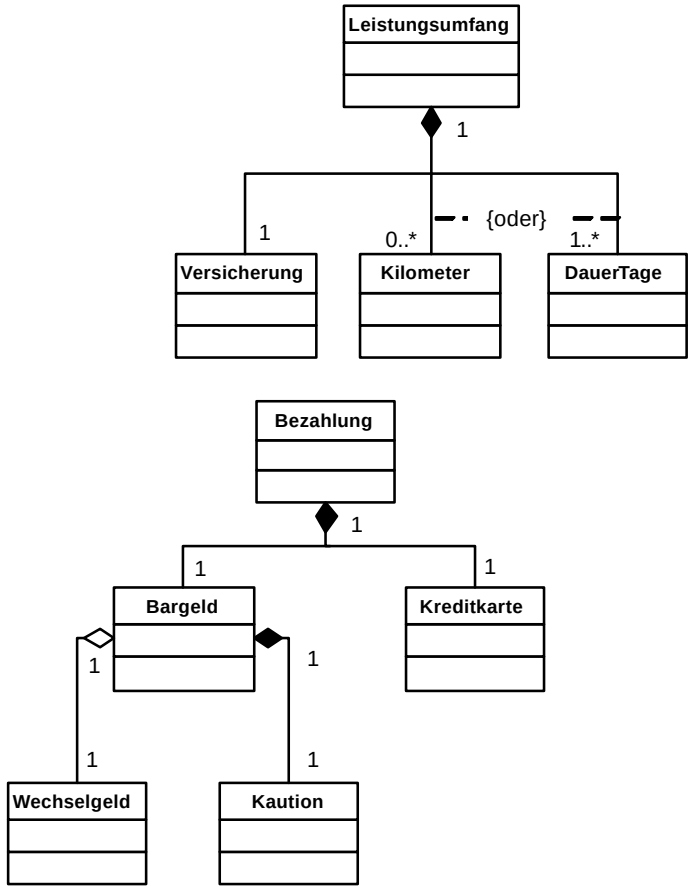

Abbildung 8.63 Aggregationen und Komposita

Klassen erstellen

Nachdem Sie abstrakte Klassen, Assoziationen, Aggregationen und Komposita erstellt haben, dürften Sie schon ein gewisses Verständnis erlangt haben, so dass Sie sich ans Werk machen können, um die einzelnen Klassen zu modellieren. Auch hier gilt, dass Sie zunächst die Klassen mit den wichtigsten Attributen und Operationen erstellen sollten. Dabei werden Sie dieses Modell immer mehr verfeinern und weitere Attribute und Operationen hinzufügen. Auch hier wollen wir mit den wichtigsten Klassen beginnen, was in diesem Fall die Personen selbst sind.

Die erste und wichtigste Person ist der Kunde. Welche wichtigen Attribute benötigen wir von ihm? Auf den ersten Blick könnte man hier folgende Attribute verwenden:

- Name
- Alter
- Nationalität
- Adresse
- Bestelldatum

Hierbei stellt sich schon die erste Design-Frage. Wenn wir hier bereits das Bestelldatum bei den Attributen verwenden, könnte man doch auch gleich den ganzen Leistungsumfang zu den Attributen des Kunden schreiben und nicht eine eigene Klasse dafür verwenden. Solche Bedenken sollte man auf jeden Fall noch vor der Erstellung des Codes ausräumen. Außerdem kann man sich auch gleich darüber Gedanken machen, ob man nicht auch noch eine Klasse `Adresse` einführt, da eigentlich alle Personen in diesem Projekt eine Adresse haben. Ansonsten müsste man die Adresse in einzelne Attribute aufteilen (Straße, Ort, PLZ etc.).

Die Operationen des Kunden lassen sich recht einfach aus dem Modell mit der Assoziation herauslesen. So sind einige der Operationen des Kunden die folgenden:

- `mietet()` – natürlich handelt es sich hier um das Mietobjekt Auto.
- `kauftHinzu()` – hier ist die Rede vom zusätzlichen Leistungsumfang, den der Kunde erwirbt.
- `bezahlt()` – der Kunde bezahlt hier entweder mit Bargeld oder mit Kreditkarte. Je nachdem, welche Art der Bezahlung gewählt wird, muss auch eine Kaution hinterlegt werden.

Somit hätten wir vorerst eine Klasse `Kunde` wie folgt modelliert (siehe Abbildung 8.64):

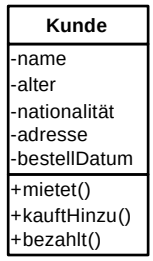

Abbildung 8.64 Die Klasse »Kunde«

Als Nächstes könnte man die abstrakte Klasse `Angestellter` modellieren. Alle Angestellten wie der Mitarbeiter an der `Rezeption`, der `Bearbeiter`, der `Mechaniker` und der `Bereitsteller` sind Unterklassen der abstrakten Klasse `Angestellter`. Zumindest haben wir dies so modelliert. Zunächst sollten Sie sich überlegen, welche Attribute alle Angestellten gemeinsam haben. Folgendes würde sich dafür vorerst anbieten:

- Name
- Adresse
- Einstelldatum
- Sozialversicherungsnummer
- Gehalt

Weitere spezifische Attribute (falls vorhanden) können Sie noch in den einzelnen Unterklassen `Rezeption`, `Bearbeiter`, `Mechaniker` und `Bereitsteller` eintragen.

Als Nächstes sollten Sie sich die einzelnen Operationen der Unterklassen überlegen. Zunächst sollte das Wichtigste und Nötigste modelliert werden. Häufig lassen sich in einer solchen Operation mehrere Vorgänge zusammenfassen. Verfeinern können Sie das Modell immer noch. Operationen für die Rezeption wären:

- `empfängtKunden()`
- `informiertBearbeiter()`

Für den Bearbeiter:

- `vermietetAuto()`
- `abrechnen()`
- `benachrichtigtBereitsteller()`

Für den Mechaniker:

- `repariert()`
- `statusReperaturMelden()`

Für den Bereitsteller:

- `bereitetAutoVor()`
- `fährtAutoVor()`
- `nimmtAutoAb()`

Modellieren Sie dies in einem UML-Diagramm, ergibt sich folgende Zusammenfassung (siehe Abbildung 8.65):

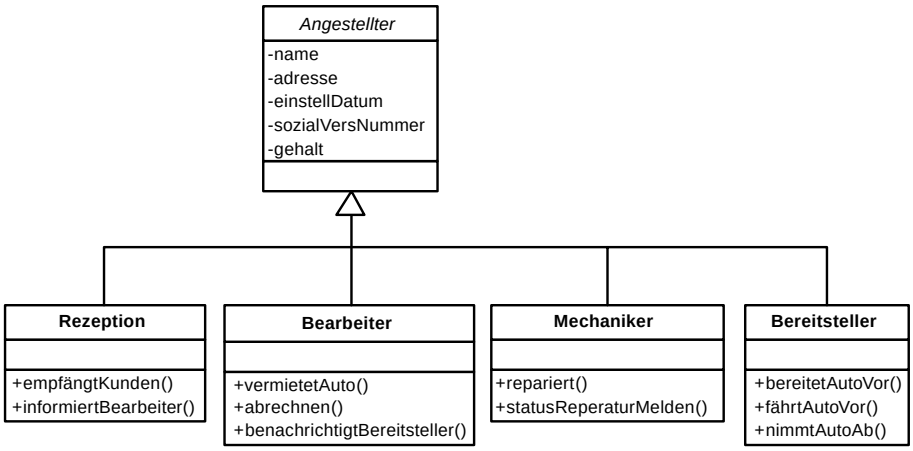

Abbildung 8.65 Die abstrakte Klasse »Angestellter« und die Unterklassen

Die Klasse `Auto` lässt sich wohl am einfachsten erstellen. Hierzu sind (für eine Firma, die Autos vermietet) folgende Attribute von Interesse:

- Marke
- Baujahr
- Benzinverbrauch auf 100 km
- Klimaanlage
- Anzahl Türen
- Airbag
- Radio/CD
- Zustand
- Zustandsbeschreibung
- Kilometerstand
- Kilometerpreis
- Preis pro Tag

Wie bereits erwähnt, ist es hierbei nicht so wichtig, wie ein Motor an oder aus geht. Daher werden auch nur Operationen ausgesucht, die für eine Autovermietung sinnvoll erscheinen. In diesem Fall wären dies:

- Preis in Kilometern berechnen
- Preis in Tagen berechnen

Als UML-Modell sieht die Klasse `Auto` folgendermaßen aus:

Abbildung 8.66 Die Klasse »Auto«

Das Gleiche können Sie zur Übung nun selbst mit der abstrakten Klasse `Vermietung` machen, wenn Sie wollen.

Zusammenfassung

Wenn Sie den Praxisteil zuvor durchgelesen haben, werden Ihnen wahrscheinlich noch viel mehr Details und Einzelheiten eingefallen sein. Einiges hätten Sie vielleicht anders gemacht. Und genau da liegt der Vorteil bei einem Projekt, wenn Sie UML verwenden. Wenn Sie einfach anfangen zu programmieren, hätten Sie zwar unzählige Klassen voll mit Attributen und Operationen, aber wenn Sie nachträglich etwas ändern oder einbauen wollen, geht viel Zeit verloren, weil Sie bei jedem eingebauten Attribut und jeder Operation überprüfen müssen, ob das Programm nach wie vor korrekt abläuft.

In der Praxis werden solche UML-Diagramme meistens von Projektleitern angelegt, so dass Sie eher selten ein Modell erstellen müssen. Aber Sie müssen verstehen können, was in diesem Diagramm modelliert wird – und dafür sind Sie nun auf jeden Fall gerüstet. Auf der anderen Seite weiß man ja nie, ob man nicht irgendwann selbst ein Projekt leiten muss.

8.5 Programmierstil

Dieses Thema in ein Programmierbuch aufzunehmen ist zwar selbstverständlich, aber nicht unbedingt einfach, weil es eben keine festen Regeln dafür gibt. Was gehört alles zum Thema »Programmierstil«, und warum sollte man sich überhaupt darüber Gedanken machen?

Wenn Sie Programme geschrieben haben, werden Sie anschließend feststellen, dass Sie mehr Zeit mit der Pflege, Aktualisierung und dem Debuggen verbringen werden als mit der Programmerstellung selbst. Gerade in der Open-Source-Zeit werden die meisten Programme auf bereits existierender Software aufgebaut. Somit wird häufig ein bereits existierender Code erweitert, verbessert oder umgeschrieben. Wenn man sich hierbei keine Gedanken über den Stil macht, wird man irgendwann Probleme haben, den Code zu verbessern. Außerdem ist nichts ärgerlicher, als viel Zeit damit zu verbringen herauszufinden, was das Stückchen Code eigentlich macht, nur weil ein fauler Programmierer keine Lust hatte, seinen Code zu dokumentieren.

Daher sollte sich jeder Programmierer hinsichtlich des Stils über folgende Dinge Gedanken machen:

- Kommentare
- Code
- Benennung
- Codeformatierung

Diese vier Dinge entscheiden (nicht nur) darüber, ob ein Programmierstil sauber ist oder nicht. Darauf soll jetzt ein wenig genauer eingegangen werden.

8.5.1 Kommentare

Was Kommentare sind, muss ich wohl nicht mehr erwähnen. Häufig wird der Rat gegeben, möglichst viel zu kommentieren, aber im Grunde ist dies veraltet und wird sowieso nicht auf Dauer eingehalten. Ein einfaches Beispiel dazu:

```
unsigned int kontoNummer;  // Kontonummer
string wohnort;            // Wohnort

// Die Klasse XY
class XY {
   //...
};
```

```
// gibt Hallo aus
cout << "Hallo" << endl;
```

Hier sind die Beschriftungen einfach fehl am Platz. Jeder sieht selbst, wofür die Variablen stehen. Wenn dies nicht so ist, dann sollte man sich vielleicht vorher Gedanken über die Bezeichner der Variablen machen. Dass die Klasse `XY` hier steht, weiß jeder C++-Programmierer von selbst. Was `cout` hier macht, erkennt jeder Anfänger und muss deshalb nicht kommentiert werden.

Leider sind viele Quellcodes voll von überflüssigen Kommentaren, so dass schnell der Überblick verloren geht. Hier gilt die goldene Regel: Kommentieren Sie nur so viel wie nötig. Dinge, die klar verständlich sind und jeder Programmierer sowieso versteht, sollten nicht kommentiert werden.

Dagegen findet man oft überhaupt keine Kommentare, wenn komplizierte Codeabschnitte auftauchen. Aber genau dann sind Kommentare nötig. Da stellt sich dann die Frage, ob der Programmierer den Codeabschnitt überhaupt verstanden hat. Ich gebe zu, ich habe auch schon häufig einen Code (ab)geschrieben, den ich nicht verstanden habe, aber ich habe mir die Mühe gemacht, diesen zu kommentieren, und als Belohnung konnte ich den Code auf diese Weise besser nachvollziehen.

Wenn Sie einen wichtigen Codeabschnitt besonders hervorheben wollen, dann können Sie dies mit einem »Kasten« tun:

```
/*****************************************
 *      _                                *
 *     / \_____________                  *
 * _ /   /\_)____________)  ACHTUNG!!!   *
 *       ()_____)                        *
 *       ()_____)                        *
 * --\___()____)                         *
 *                                       *
 *   Wichtige Kommentare .......         *
 *****************************************/
```

Natürlich müssen Sie nicht gleich, wie hier, ein ASCII-Bild verwenden, doch wenn es darum geht, einen Kommentar aus der Masse hervorzuheben, kann man auch zu ungewöhnlichen Mitteln greifen und dem Kommentar eine persönliche Note verleihen.

Ein weiteres Problem beim Kommentieren von Quellcode entsteht, wenn ein Kommentar nichts aussagt:

```
int max_wert;   // maximaler Wert
```

In einem Programm mit mehreren Modulen und zigtausend Zeilen »freut« man sich über einen solchen Kommentar. Zumindest sollte man hier hinschreiben, für was dieser maximale Wert genau ist. Besser wäre also Folgendes:

```
int max_wert;   // maximale Anzahl von Dateien,
                // die geöffnet werden können
```

8.5.2 Code

Manche Programmierer neigen dazu, besonders klug erscheinen zu wollen, indem Sie einen Code schreiben, der schwer zu verstehen ist. Ein Beispiel dazu:

```
const char *j[] = {"nein", "ja"};
...
cout << j[!!(caps & CDC_CD_R)];
```

Mit `j[!!(caps & CDC_CD_R)]` werden die meisten Schwierigkeiten haben. Hierbei wird überprüft, ob in der Variablen `caps` das Flag `CDC_CD_R` gesetzt ist (unabhängig davon, was hier die Variable und das Flag für eine Bedeutung haben). Was für einen C-Programmierer an dieser Stelle eine Optimierung des Codes darstellt, ist für den »normalen« Programmierer kaum mehr nachzuvollziehen. Verwendet man den ternären Operator `?:`, dann sieht dieses Konstrukt schon einleuchtender aus:

```
cout << ((caps & CDC_CD_R) ?"nein" :"ja");
```

Aber auch der ternäre Operator ist nicht unbedingt sehr beliebt, wenn es um die Lesbarkeit von Programmen geht. Daher kann man diesen Codeausschnitt noch mit einer schlichten `if`-Anweisung vereinfachen:

```
if( caps & CDC_CD_R ) {
   cout << "nein";
}
else {
   cout << "ja";
}
```

Hinweis **[«]**

Hierbei stellt sich immer die Frage, ob der Code an dieser Stelle unbedingt optimiert werden muss oder ob man es auch bei einem leichter lesbaren Code belassen kann.

Des Weiteren optimieren viele Compiler mittlerweile den Code bereits selbst und erzeugen aus allen drei Versionen denselben Maschinencode.

8.5.3 Benennung

Neben ordentlichen Kommentaren ist auch die Benennung von Variablen, Klassen, Funktionen etc. von Bedeutung. Es gibt hierbei keine festen Regeln (abgesehen von den Namen der Bezeichner, siehe Abschnitt 1.3.1, »Bezeichner«). Bei einem Projekt mit mehreren Programmierern sollte man dies sowieso vorher absprechen.

Gewöhnlich verwendet man für Variablen Kleinbuchstaben. Mittlerweile findet man aber häufig Beispiele, die Groß- und Kleinschreibung mischen, aber der erste Buchstabe wird meistens kleingeschrieben:

```
int maxgewicht;
int maxGewicht;
```

Über den Namen sollte man sich ebenfalls Gedanken machen. Ein schlechtes Beispiel ist

```
int max1, max2, max3;
```

Man sollte schon beim Bezeichner mehr Informationen angeben:

```
int maxVolumen, maxGewicht, maxHerstellung;
```

Außerdem sollte man »ganze« Variablen-Bezeichner verwenden und den Einsatz von Ein-Buchstaben-Bezeichnern vermeiden. Eine Ausnahme stellen hier die Schleifenvariablen dar, die recht häufig mit `i`, `j` etc. bezeichnet werden.

[»] **Hinweis**

Bedenken Sie bei der Namensvergabe auch, dass es Programmierer gibt, die vielleicht Ihre Muttersprache nicht beherrschen. Sofern Sie also vorhaben, sämtliche Variablen-Bezeichner in Ihrer Sprache anzugeben, sollten Sie überdenken, ob der Code vielleicht auch international interessant sein soll. Dann sollten Sie die englische Sprache verwenden.

8.5.4 Codeformatierung

Um den Code leichter, übersichtlicher und verständlicher zu machen, werden gewöhnlich bei einem neuen Anweisungsblock oder bei Bedingungen Einrückungen vorgenommen. Hierbei gibt es mehrere Möglichkeiten. Die im Buch verwendete Codeformatierung ist darauf ausgerichtet, möglichst wenig Platz zu verschwenden:

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

Ein ebenfalls übersichtlicher Stil ist es, die sich öffnende geschweifte Klammerung in eine Extrazeile zu schreiben:

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

Viele Editoren (bzw. Entwicklungsumgebungen) übernehmen außerdem die Einrückung in einem Anweisungsblock selbst. Hierbei bleibt es wieder dem Programmierer überlassen, ob er hier zwei, vier oder noch mehr Leerzeichen (bzw. den Tabulator) dazu verwendet.

Bei Projekten mit mehreren Programmierern sollte man sich außerdem auf dieselbe Codeformatierung einigen.

Hinweis [«]

Welchen Stil man auch verwendet, es sollte immer die Regel *eine Anweisung pro Zeile* gelten. Wenn ein Fehler auftritt, ist es so einfacher, diesen zu lokalisieren.

Für die Übersichtlichkeit des Codes kann es sinnvoll sein, gelegentlich Leerzeilen einzufügen. Folgende Zeilen

```
ivalTmp = ival1;
ival1 = ival2 ;
ival2 = ivalTmp ;
ival1Copy = ival1 ;
ival2Copy = ival2;
```

können folgendermaßen übersichtlicher dargestellt werden:

```
// Werte ival1 und ival2 tauschen
ivalTmp = ival1;
ival1 = ival2 ;
ival2 = ivalTmp ;

// Kopie der beiden Werte erzeugen
ival1Copy = ival1 ;
ival2Copy = ival2;
```

8.5.5 Zusammenfassung

Natürlich gibt es zum Thema »Programmierstil« noch viel zu sagen. Aber wenn Sie nur die Kleinigkeiten beachten, die hier erwähnt wurden, haben Sie schon eine Menge für Ihren eigenen Stil getan.

8.6 Entwicklungsstufen von Software

Nach der ersten Auflage dieses Buches wurden einige Fragen bezüglich der Planung und Erstellung von Software gestellt. Zu diesem Thema gibt es mittlerweile ganze Bücher, und es empfiehlt sich auch, sich solche Literatur früher oder später einmal anzuschaffen. Trotzdem will ich die Gelegenheit hier nutzen, um Ihnen ein paar Ratschläge mit auf den Weg zu geben.

Aus eigener Erfahrung weiß ich, dass das Entwickeln von Programmen eine ziemlich komplexe, langwierige und häufig auch frustrierende Angelegenheit sein kann. Wer vielleicht einige Jahre in C programmiert hat (wie ich), der ist garantiert in die diese Falle gelaufen: In C war man es gewohnt, einfach draufloszuprogrammieren. Macht man dies in C++, verwendet man es oft im Grunde nur als besseres C. Benutzen Sie dann dabei Klassen und die mächtigen Techniken von C++, wird das objektorientierte Design häufig aufgeweicht. Man setzt dann einfach unnötige Dinge auf `public` oder verwendet unnötige globale Funktionen oder gar Variablen. Was am Ende dann herauskommt, ist alles andere als sauberer Code.

Natürlich hängt der ganze Prozess auch vom Umfang des Projekts ab. Bei kleineren Projekten (weniger als 10.000 Zeilen) mag dieses Drauflosprogrammieren vielleicht noch funktionieren, aber mit steigendem Umfang und Klassen wird die Sache immer komplexer. Sind dann noch mehrere Entwickler an der Arbeit, ist eine umfassende vorherige Planung der Software unbedingt Pflicht.

Aus der Erfahrung weiß ich mittlerweile, dass Sie sich mit einer sorgfältigen und guten Planung eine Menge Codezeilen (bis zu 50 %) sparen können. Die schmerzliche Erfahrung musste ich machen, als ich bei einem meiner ersten Projekte einfach drauflosprogrammiert habe. Bei einem Relaunch des Codes haben mir Experten den Code total zerlegt, man kann sagen, neu geschrieben. Der anschließende Code war um mehr als die Hälfte schlanker und hat nur etwa ein Viertel der Zeit für die Fertigstellung gebraucht, die ich darauf verwendet hatte. Zugegeben, diese Erfahrung zu machen ist recht unangenehm fürs Geschäft, hat aber auch eine positive Seite: Ich habe dadurch gelernt, dass Programmieren nicht einfach nur Programmieren ist, sondern dass es hier weitaus mehr zu beachten gibt.

[»] **Hinweis**

Auch wenn viel über die Themen Entwicklung, Design, Testen und Debuggen geschrieben wird, muss man doch immer wieder betonen, dass es keinen ultimativen, richtigen Weg dafür gibt. Daher sei auch das von mir hier Beschriebene nur als Empfehlung aus meiner Sicht zu verstehen. Ich will damit erreichen, dass Sie als Leser nicht gleich bei Ihrem ersten Projekt das Gefühl haben, nicht mehr zu wissen, wo Ihnen der Kopf steht – so, wie es mir und wohl vielen anderen am Anfang ergangen ist.

Abbildung 8.67 soll grob die unterschiedlichen Stufen des Entwicklungsprozesses darstellen.

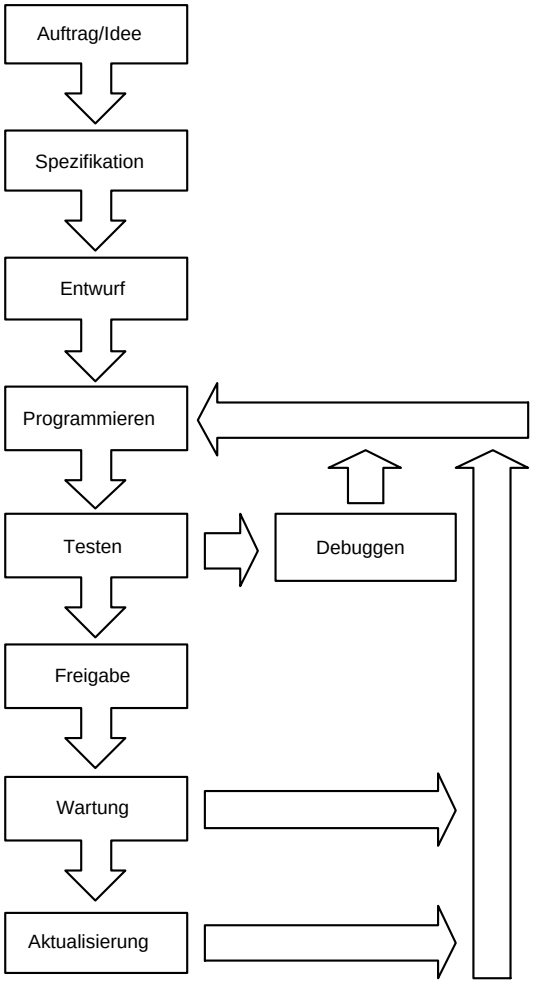

Abbildung 8.67 Entwicklungsstufen einer Software

8.6.1 Auftrag bzw. Idee

Bevor Sie ein Projekt starten können, brauchen Sie natürlich einen Grund, weshalb Sie das tun wollen. Die meisten Programmiereinsteiger werden sich wohl zunächst mit einem eigenen Projekt einen Namen machen und Erfahrung sammeln wollen. Andere wiederum haben das Glück, den Auftrag zu erhalten, ein Softwareprojekt zu entwickeln. Am besten wäre es natürlich, wenn Sie an einem

Projekt mitwirken könnten, an dem mehrere Personen mit sehr viel Erfahrung beteiligt sind. Dabei könnten Sie am meisten lernen.

Aber egal, ob Sie jetzt Hobby-, freiberuflicher oder hauptberuflicher Programmierer sind, zunächst muss man wissen, was man machen will. Gehen Sie auf Foren und trommeln Sie mehrere Programmierer zusammen, oder fragen Sie, was andere von dem Projekt halten, das Ihnen im Kopf rumschwirrt. Lesen Sie die Meinungen der anderen und bilden Sie sich daraus Ihre eigene. Ein häufiger Fehler vieler Hobbyprogrammierer ist beispielsweise, dass sie Projekte planen, dessen Ausmaß ihnen nicht bekannt ist. Meine Empfehlung deshalb: Suchen Sie sich zunächst kleinere, realistische Projekte aus – vor allem, wenn Sie allein programmieren. Es gibt mittlerweile mehr unvollendete als fertiggestellte Projekte.

Natürlich träumt jeder auch einmal davon, ein wenig Geld mit der Entwicklung von Programmen zu verdienen. Gut, melden Sie ein Gewerbe an, und steigen Sie als freiberuflicher Programmierer ein. Machen Sie Werbung in lokalen Tageszeitungen, Zeitschriften und natürlich auch mit einem ordentlichen Internetauftritt. Hierbei sollten Sie aber wirklich sicher sein, was Sie tun. Einen Auftraggeber findet man häufig recht schnell, aber die Verantwortung und der Zeitaufwand sind enorm. Ganz nebenbei muss man ja auch auf dem Laufenden bleiben, was die IT-Welt anbetrifft. Mit Standard-C++ allein werden Sie wohl kaum ein Softwareprojekt entwickeln. Häufig muss man noch eine GUI für den Kunden aufsetzen und die Daten in einer Datenbank speichern. Natürlich will der Kunde diese Datenbank auch einfach administrieren können. Sie müssen also auch viel über vorhandene Schnittstellen, Bibliotheken und Frameworks wissen.

8.6.2 Spezifikation und Anforderung

Jetzt, nachdem Sie eine Idee oder einen Auftrag erhalten haben, wissen Sie zwar schon in etwa, wie das Programm aussehen sollte und was es macht, aber nun müssen Sie die Anforderungen und Spezifikationen zusammenstellen.

Was genau wird alles für das Programm benötigt? Sind fertige Bibliotheken oder Frameworks für die Lösungen vorhanden, oder müssen Sie neue schreiben? Wenn Sie eine neue Schnittstelle schreiben müssen, brauchen Sie auch eine Beschreibung dazu. Müssen Sie beispielsweise Daten an ein externes Programm über das Netzwerk verschicken, benötigen Sie die Spezifikation des externen Programms, wie die Daten auszusehen haben. Sie werden dabei wahrscheinlich nicht um das Lesen weiterer Dokumente herumkommen.

Oder muss das Programm auf mehreren Systemen laufen und somit portabel sein? Gibt es entsprechende Bibliotheken und Frameworks auch für diese Sys-

teme, und wie sieht es mit den Lizenzen der Bibliotheken aus? Welche Extrakosten fallen dabei an, und gibt es kostenlose Alternativen?

Nachdem Sie die Anforderungen zusammengestellt haben, müssen Sie die Spezifikation für das Programm erstellen. Das bedeutet, Sie schreiben eine vorläufige Spezifikation des Projekts, was das Programm macht – am besten gleich so genau wie möglich. Wenn Sie die Spezifikation erstellt haben, sollten Sie diese zunächst dem Kunden geben, um spätere Missverständnisse zu vermeiden. Fragen Sie den Kunden, ob ihm noch etwas dazu einfällt, und verfeinern Sie so nach und nach die Spezifikation. Erst wenn beide Parteien zufrieden sind, können Sie zum nächsten Schritt übergehen.

Hinweis [«]

Wenn Ihr Projekt zunächst ein reines Ein-Mann-Hobbyprojekt ist, können Sie Ihre Spezifikation des Programms auch von einem Forum beurteilen lassen.

Diese Spezifikation stellt natürlich noch nicht das endgültige Ergebnis dar und wird im Verlauf der Entwicklung immer weiter verbessert bzw. verfeinert. Die Verbesserungen sollten aber immer in der Spezifikation festgehalten und mit dem Kunden besprochen werden. Am Ende Ihres Softwareprojekts erhalten Sie dadurch eine vollständige Beschreibung, was das Programm tut, und können daraus gegebenenfalls auch eine Dokumentation des Programms erstellen.

8.6.3 Entwurf (Design)

Der Vorgang der Entwurfserstellung ist der wichtigste in der ganzen Kette der Softwareentwicklung. Ein guter Entwurf erspart dem Programmierer eine Menge Zeit und Nerven. Ein erster Entwurf sieht ein grobes Design des Programms vor. Hierbei legen Sie die Klassendefinitionen, Operationen, Abhängigkeiten, Schnittstellen, Klassenhierarchien usw. fest.

Bei mehreren Programmierern im Projekt wird hierbei auch gleich die Arbeit in einzelne Module aufgeteilt. Es sollte auch festgelegt werden, welches Format die Daten haben oder welche Algorithmen verwendet werden. Natürlich umfasst dies auch die Planung, wie externe Bibliotheken oder Frameworks in das Projekt eingebunden werden.

Ein gutes Hilfsmittel für den Entwurf wäre UML, wie dies in Abschnitt 8.4, »UML«, bereits näher beschrieben wurde. Nehmen Sie sich auf jeden Fall Zeit mit dem Entwurf der Software. Ein nachträgliches Einpflücken einer Klasse oder Methode kann einen enormen Aufwand bedeuten. Aus dem Entwurf heraus kann der Programmierer sein Grundgerüst erstellen. Man sollte hieraus also Klassen, Methoden, Operationen und Abhängigkeiten (Vererbungs- und Verwen-

dungsbeziehungen) erkennen können. Überdenken Sie dabei jedes Detail, jeden Parameter – oder denken Sie hierbei wirklich objektorientiert. Verwenden Sie lieber mehrere kleine Klassen als eine große.

Wenn der erste Entwurf fertiggestellt ist, überarbeiten Sie ihn nochmals, verfeinern Sie den alten Entwurf. Überlegen Sie immer, wo man sich die Arbeit einfacher machen kann und nicht komplexer. Wie Sie in Abschnitt 8.4, »UML«, schon gesehen haben, ist es dabei sogar möglich, dass ein Laie versteht, was das Programm machen soll. So können Sie den Entwurf auch gerne dem Kunden vorlegen und nochmals verfeinern. Damit können Sie sicherstellen, dass Sie den Kunden verstanden haben, und der Kunde wiederum kann sein Programm früher verwenden, weil die Herstellungszeit durch eine gute Planung erheblich reduziert wird.

8.6.4 Programmieren (Codieren)

Jetzt kommt der eigentliche Akt, für den Sie das Programmieren gelernt haben. Hierzu benötigen Sie Ihren Entwurf. Sie erstellen zunächst das Grundgerüst mit allen Modulen, Klassen, Methoden, Funktionen usw.

Sparen Sie nicht an Kommentaren, was wofür beim Grundgerüst verwendet wird. Wenn Sie das Grundgerüst erstellt haben, können Sie anfangen, den Code zu schreiben. Beim Erstellen des Quellcodes stellt sich heraus, wie gut der Entwurf des Programms geworden ist. Versuchen Sie zunächst, nur den Entwurf zu befolgen und nicht schon jetzt Neuerungen und Verbesserungen einzubringen. Notieren Sie sich gegebenenfalls die Verbesserungen, und warten Sie, bis die Zeit reif ist. Gerade wenn man in der Gruppe programmiert, können solche Alleingänge fatale Folgen haben. Machen Sie nicht unnötig viele Baustellen auf, das bremst den Arbeitsablauf.

8.6.5 Testen und Debuggen

Wenn das Programm nach dem Entwurf fertiggestellt ist und sich problemlos kompilieren lässt, können Sie mit dem Testen beginnen. Die Testphase ist eine sehr wichtige Phase, um das Programm auf Herz und Nieren zu prüfen. Probieren Sie zunächst den normalen Ablauf des Programms. Testen Sie jede einzelne Funktion. Wenn dies funktioniert, machen Sie absichtlich Fehler bei der Eingabe, geben Sie falsche Werte ein. Das Programm muss auch dagegen gesichert sein. Anwender geben gerne mal falsche Werte ein. Nimmt das Programm falsche Werte ohne Probleme entgegen, oder stürzt es dabei ab? Auf jeden Fall müssen Sie etwas dagegen unternehmen. Ändern Sie dies, und testen Sie weiter.

Schwieriger wird es, wenn das Programm unerwartet abstürzt. Dann wird die Suche nach dem Fehler häufig schwieriger. Versuchen Sie, den Vorgang von zuvor nochmals auszuführen, um das Abstürzen zu bewirken. Wenn Sie den Absturz lokalisiert haben, müssen Sie diesen auch im Code suchen. Hierbei behelfe ich mir immer mit einem einfachen Trick und schreibe in der Zeile, in der ich den Fehler vermute, eine Debug-Meldung auf die Konsole. Falls dies nicht möglich ist, verwenden Sie eben eine andere Form der Ausgabe. Bei einem GUI-Framework beispielsweise könnten Sie eine Nachrichtenbox anzeigen lassen.

Wird die Debug-Meldung angezeigt, schreiben Sie diese eine Zeile tiefer rein und testen erneut, so lange, bis die Debug-Meldung nicht mehr angezeigt wird. Jetzt sollte der Fehler zumindest lokalisiert sein.

Hinweis [«]

Natürlich setzt man für solche Zwecke auch den Debugger ein. Aber es hat sich herausgestellt, dass viele Fehler häufig kleinerer Natur sind und einem erfahrenen Programmierer gleich ins Auge stechen. Also sollte man nicht gleich mit Kanonen auf Spatzen schießen.

Das Wichtigste beim Testen ist jedoch, sich immer wieder in den Anwender zu versetzen. Was könnte dieser falsch eingeben? Dies ist übrigens ein häufiger Fehler, den Programmierer machen – sie verlassen sich darauf, dass der Anwender schon das Richtige eingeben wird.

8.6.6 Freigabe (Release)

Erst nach ausgiebigen Tests sollten Sie das Programm freigeben. Dabei bedeutet eine Freigabe nicht, dass keine Fehler mehr vorhanden sind. Es bedeutet lediglich, dass Ihnen keine Fehler mehr bekannt sind. Häufig ist man allerdings leider auch ein wenig unter Zeitdruck und muss eine Software bis zu einem bestimmten Zeitpunkt freigeben. Versuchen Sie zumindest, die Fehler bis dahin zumutbar niedrig zu halten und bei späteren Aktualisierungen zu beheben.

8.6.7 Wartung

Der Begriff der Wartung lässt sich in der Softwareentwicklung nicht so einfach mit der Wartung eines Autos vergleichen, sondern eher mit einer Rückrufaktion der Autoindustrie, wenn man so will. Bei der Wartung von Software werden Fehler, die nach der Freigabe aufgetaucht sind, behoben. Wenn Sie denken, dass dies selten der Fall ist, dann seien Sie gewarnt. Ein Programm wird niemals von Anfang an perfekt sein. Das passiert den besten Programmierern. Natürlich bedeutet eine Wartung auch wieder Programmieren, Testen und Debuggen.

8.6.8 Aktualisierung (Update)

Wenn das Programm länger läuft, werden dem Anwender viele Kleinigkeiten auffallen, was man verbessern oder ändern könnte. Vielleicht fällt Ihnen selbst der ein oder andere Algorithmus ein, um das Programm schneller und stabiler zu machen. Sprechen Sie hierbei mit dem Kunden bzw. den Anwendern, die das Programm längere Zeit eingesetzt haben. Eine solche Aktualisierung bedeutet natürlich auch wieder, dass Sie die Neuerungen in die Spezifikation aufnehmen.

Gegebenenfalls muss dann auch der Entwurf erneuert bzw. überarbeitet werden. Hierbei muss quasi der ganze Entwicklungsprozess nochmals durchlaufen werden. Wenn Sie sich wirklich einen guten Namen machen wollen, empfehle ich Ihnen, regelmäßige Updates für das Programm zu liefern. Häufig können Sie dem Kunden dabei neue Features für das Programm anbieten, mit denen er sein Programm noch einfacher und flexibler bedienen kann. Ihr Kunde erhält dabei ein noch besseres Programm, und bei Ihnen klingelt wieder die Kasse. Natürlich hängt dies immer von den Vereinbarungen ab, die Sie mit dem Kunden getroffen haben.

8.7 Boost

Der Begriff *Boost* umfasst eine Sammlung von freien C++-Bibliotheken. Diese Bibliotheken finden eine recht breite Unterstützung und werden mittlerweile in vielen C++ Projekten eingesetzt. Das Projekt Boost selbst wurde von den Mitgliedern des Standardisierungskomitees für C++ gegründet. Darin werden Vorschläge für die Erweiterung von C++ gesammelt – und vor allem auch getestet mit der Absicht, dass der ausgereifte Teil davon in den C++-Standard aufgenommen wird. Zehn der mittlerweile 54 Bibliotheken sind bereits sichere Kandidaten für die Aufnahme in den zukünftigen C++-Standard. Viele Bibliotheken von Boost sind im Grunde fortgeschrittene Anwendungen der C++-Templates (also generische Programmierung bzw. Metaprogrammierung).

Mit C++ und STL haben Sie schon ein mächtiges Werkzeug – aber häufig wünscht man sich noch weitere Möglichkeiten, wie reguläre Ausdrücke, Arbeiten mit Verzeichnissen (bzw. Filesystem) oder dem Multithreading. Zwar gibt es für all diese Dinge eine Bibliothek (oder man kann auch selbst eine erstellen), aber die Suche danach kann man sich sparen. Hierbei kommt dann auch noch die Arbeit dazu, diese Bibliothek ins Projekt zu implementieren und zu testen. Dafür bietet sich Boost an – diese Bibliotheken sind getestet, ausgereift und sehr gut dokumentiert.

Und – last but not least – wenn man Dinge wie das Arbeiten mit Verzeichnissen oder Multithreading portieren muss, dann stößt man schnell an die Grenzen der Portabilität, so dass man zum Teil gezwungen ist, zwei Klassen dafür zu schreiben.

Die Liste der Bibliotheken zu Boost ist mittlerweile gewaltig, aber immer noch sehr übersichtlich. Aufteilen lassen sich diese einzelnen Bibliotheken in folgende Themen:

- Zeichenketten- und Textverarbeitung
- Container
- Iteratoren
- Algorithmen
- Funktionsobjekte
- Generische Programmierung
- Template-Metaprogrammierung
- Präprozessor-Metaprogrammierung
- Multithread-Programmierung
- Mathematisches und Numerik
- Korrektheit und Testen (Debugging)
- Datenstrukturen
- Input/Output
- Inter-Language-Support
- Memory (Speicherverwaltung)
- Parsen
- Programmierschnittstellen
- Verschiedenes
- Broken compiler workarounds

Ursprünglich war geplant, an dieser Stelle etwas ausführlicher auf die einzelnen Bibliotheken von Boost einzugehen. Aber leider stehen einem als Autor nicht unbegrenzt viele Seiten zur Verfügung, so dass ich nur als »Kostprobe« zu dieser Bibliothek auf die regulären Ausdrücke mit der Bibliothek `Boost.Regex` eingehen will – eine sehr interessante Bibliothek, wie ich finde, die man immer gebrauchen kann.

[»] **Hinweis**

Wenn Sie weitere Informationen zu einigen dieser Bibliotheken benötigen, finden Sie auf der Webseite *http://www.boost.org* eine sehr umfangreiche Dokumentation. Außerdem stehen dort die einzelnen Bibliotheken zum Download zur Verfügung mitsamt der Installationsanleitung auf den einzelnen Plattformen bzw. Compiler.

8.7.1 Boost.Regex (reguläre Ausdrücke)

Die regulären Ausdrücke werden verwendet, um Texte zu durchsuchen bzw. zu formatieren. Für Entwickler, die in Perl programmieren oder die UNIX-Tools awk, sed oder grep verwenden, sind reguläre Ausdrücke alltäglich.

[»] **Hinweis**

Mehr zu den Tools awk, sed und grep und den regulären Ausdrücken finden Sie in meinem Buch »Shell-Programmierung«, das im selben Verlag wie dieses Buch erschienen ist.

Wer bisher reguläre Ausdrücke in seinem Programm verwenden wollte, hat wohl auf eine POSIX C API mit den Funktionen `regcomp()` und `regexec()` zurückgegriffen (gibt es nicht für MS Visual C++). Allerdings hatte diese C API auch einige Einschränkungen. So konnte man damit nicht mit Wide-Char-Strings (`wchar_t`) arbeiten – auch ein einfaches Suchen und Ersetzen war hiermit nicht (direkt) möglich. Mit `Boost.Regex` ist dies jedoch möglich. Wer außerdem schon Erfahrungen mit den regulären Ausdrücken gesammelt hat und die POSIX-konforme Syntax kennt, kann sich freuen, weil `regex` von Boost ebenfalls POSIX-konform ist. Was ein weiteres Plus für `regex` (auch `regex++` genannt) von Boost darstellt, ist eine komfortable Schnittstelle, die die Verwendung von regulären Ausdrücken fast zu einem Kinderspiel macht (wenn man sich damit auskennt).

[»] **Hinweis**

Um reguläre Ausdrücke von Boost zu verwenden, muss natürlich die erforderliche Bibliothek vorhanden sein. Häufig setzt dies voraus, dass Sie die entsprechende Bibliothek auf dem entsprechenden Computer übersetzen müssen. Wer hier den Borland-, MS- oder GCC-Compiler verwendet, für den dürfte dies keine allzu große Hürde darstellen. Hierzu möchte ich Sie auf die Webseite *www.boost.org* verweisen. Dort finden Sie die entsprechende Dokumentation für den entsprechenden Compiler und die Information, wie Sie `regex++` installieren können.

[»] **Hinweis**

Vielleicht sollte auch erwähnt werden, dass die regulären Ausdrücke von Boost bereits in größeren Projekten wie beispielsweise von SAP verwendet werden, nämlich in SAPs Application Server NetWeaver (mySAP), der eine eigene Sprache namens ABAP hat. Der Interpreter wurde hierbei mit Boosts `regex` implementiert.

Reguläre Ausdrücke (eine kurze Einführung)

Reguläre Ausdrücke (engl.: *regular expression*) sind eine leistungsfähige formale Sprache, mit der sich eine bestimmte (Unter-)Menge von Zeichenketten beschreiben lässt. Es muss hierbei allerdings gleich erwähnt werden, dass reguläre Ausdrücke kein Tool und keine Sammlung von Funktionen sind, die von einem Betriebssystem abhängig sind, sondern es handelt sich um eine echte Sprache mit einer formalen Grammatik, in der jeder Ausdruck eine präzise Bedeutung hat.

Das Einsatzgebiet von regulären Ausdrücken ist vielseitig, sie werden von sehr vielen Texteditoren und Programmen verwendet. Meistens benutzt man reguläre Ausdrücke, um bestimmte Muster zu suchen und diese dann durch etwas anderes zu ersetzen. In der Linux/UNIX-Welt werden reguläre Ausdrücke vorwiegend bei Programmen wie grep, sed und awk oder den Texteditoren `vi` und `Emacs` verwendet. Aber auch viele Programmiersprachen wie u.a. Perl, Java, Python, Tcl, PHP oder Ruby bieten reguläre Ausdrücke an.

Die Entstehungsgeschichte der regulären Ausdrücke ist recht schnell erzählt. Den Anfang hat ein Mathematiker und Logiker, Stephen Kleene, gemacht. Er gilt übrigens auch als Mitbegründer der theoretischen Informatik, besonders der hier behandelten formalen Sprachen und der Automatentheorie. Stephen Kleene verwendete hierbei eine Notation, die er selbst *reguläre Menge* nannte. Später verwendete dann Ken Thompson (der Miterfinder der Programmiersprache C) diese Notationen für eine Vorgänger-Version des UNIX-Editors `ed` und für das Werkzeug `grep`. Nach der Fertigstellung von `grep` wurden die regulären Ausdrücke in sehr vielen Programmen implementiert. Viele davon benutzen die mittlerweile sehr bekannte Bibliothek `regex` von Henry Spencer.

Hinweis [«]

Sofern Sie Erweiterungen wie Rückwärtsreferenzen verwenden wollen, sei hierzu Perl empfohlen, weil grep das nicht leisten kann. Wobei inzwischen ja verschiedene regexes (POSIX-RE, Extended-RE und pcre) unterschieden werden. Die Unterschiede sind in den Manuals regex und perlre zu finden. Ein Großteil der Scriptsprachen und Programme stützt sich auf pcre (*Perl Compatible Regualar Expression*), die mittlerweile als die leistungsfähigste gilt.

Elemente für reguläre Ausdrücke (POSIX-RE)

Vorwiegend werden reguläre Ausdrücke dazu verwendet, bestimmte Zeichenketten in einer Menge von Zeichen zu suchen und zu finden. Die nun folgende Beschreibung ist eine gewöhnlich sehr häufig verwendete Konvention, die von fast allen Programmen, die reguläre Ausdrücke verwenden, so eingesetzt wird. Normalerweise wird dabei ein regulärer Ausdruck aus den Zeichen des Alphabets in Kombination mit den Metazeichen (die hier gleich vorgestellt werden) gebildet.

Zeichen-Literale

Als *Zeichen-Literale* bezeichnet man die Zeichen, die wörtlich übereinstimmen müssen. Diese werden im regulären Ausdruck direkt (als Wort) notiert. Hierbei besteht je nach System auch die Möglichkeit, alles in hexadezimaler oder oktaler Form anzugeben.

Beliebiges Zeichen

Für ein einzelnes beliebiges Zeichen verwendet man einen Punkt. Dieser Punkt kann dann für ein fast beliebiges Zeichen stehen.

Zeichenauswahl

Die Zeichenauswahl kennen Sie ebenfalls bereits aus der Shell mit den eckigen Klammern `[auswahl]` – siehe Abschnitt 1.10.6, »Inline-Funktionen«. Alles, was Sie in den eckigen Klammern schreiben, gilt dann exakt für ein Zeichen aus dieser Auswahl. Beispielsweise steht `[axz]` für eines der Zeichen »a«, »x« oder »z«. Dies lässt sich natürlich auch in Bereiche aufteilen. Bei der Angabe von `[2-7]` besteht der Bereich zum Beispiel aus den Ziffern 2 bis 7. Mit dem Zeichen `^` innerhalb der Zeichenauswahl, können Sie auch Zeichen ausschließen. Mit `[^a-f]` zum Beispiel schließen Sie die Zeichen »a«, »b«, »c«, »d«, »e« oder »f« aus.

Vordefinierte Zeichenklassen

Manche Implementierungen von regulären Ausdrücken bieten auch vordefinierte Zeichenklassen an. Sofern Sie keine vordefinierten Zeichenklassen finden, lässt sich dies auch selbst durch eine Zeichenauswahl in eckigen Klammern beschreiben. Die vordefinierten Zeichenklassen sind letztendlich auch nur eine Kurzform der Zeichenklassen. Hier einige bekannte vordefinierte Zeichenklassen:

Vordefiniert	Bedeutung	Selbstdefiniert
`\d`	eine Zahl	`[0-9]`
`\D`	keine Zahl	`[^0-9]`
`\w`	ein Buchstabe, eine Zahl oder der Unterstrich	`[a-zA-Z_0-9]`
`\W`	kein Buchstabe, keine Zahl und kein Unterstrich	`[^a-zA-Z_0-9]`
`\s`	Whitespace-Zeichen	`[ \f\n\r\t\v]`
`\S`	alle Zeichen außer Whitespace-Zeichen	`[^\f\n\r\t\v]`

Tabelle 8.6 Vordefinierte Zeichenklassen

Quantifizierer

Als *Quantifizierer* bzw. *Quantoren* bezeichnet man Elemente, die es erlauben, den vorherigen Ausdruck in unterschiedlicher Vielfachheit in einer Zeichenkette zuzulassen:

Quantifizierer	Bedeutung
`?`	Der Ausdruck, der vorangeht, ist optional: Er kann einmal vorkommen, muss aber nicht. Der Ausdruck kommt entweder null- oder einmal vor.
`+`	Der Ausdruck muss mindestens einmal vorkommen, darf aber auch mehrmals vorhanden sein.
`*`	Der Ausdruck darf beliebig oft oder auch gar nicht vorkommen.
`{min,}`	Der vorangehende Ausdruck muss mindestens *min*-mal vorkommen.
`{min,max}`	Der vorangehende Ausdruck muss mindestens *min*-mal vorkommen, darf aber nicht mehr als *max*-mal vorkommen.
`{n}`	Der voranstehende Ausdruck muss genau *n*-mal vorkommen.

Tabelle 8.7 Quantifizierer

Gruppierung

Ausdrücke können auch zwischen runden Klammern gruppiert werden. Einige Tools speichern diese Gruppierung ab und ermöglichen so eine Wiederverwendung im regulären Ausdruck bzw. in der Textersetzung über `\1`. Es lassen sich hiermit bis zu neun Muster abspeichern (`\1`, `\2`, ...,`\9`). Zum Beispiel würde man mit

```
s/\(string1\) \(string2\) \(string3\)/\3 \2 \1/g
```

erreichen, dass in einer Textdatei alle Vorkommen von

```
string1 string2 string3
```

umgeändert würden in

```
string3 string2 string1
```

`\1` bezieht sich also immer auf das erste Klammernpaar, `\2` auf das zweite usw.

Alternativen

Selbstverständlich lassen sich hierbei auch Alternativen definieren. Hierfür wird das Zeichen `|` verwendet:

```
(asdf|ASDF)
```

bedeutet, dass hier nach »asdf« oder »ASDF« gesucht wird, nicht aber nach »AsDf« oder »asdF«.

Sonderzeichen

Da viele Tools direkt auf Textdateien zugreifen, finden Sie gewöhnlich noch folgende Sonderzeichen definiert:

Sonderzeichen	Bedeutung
^	Steht für den Zeilenanfang.
$	Steht für das Zeilenende.
\b	Steht für die leere Zeichenkette am Wortanfang oder am Wortende.
\B	Steht für die leere Zeichenkette, die nicht den Anfang oder das Ende eines Wortes bildet.
\<	Steht für die leere Zeichenkette am Wortanfang.
\>	Steht für die leere Zeichenkette am Wortende.
\d	Ziffer
\D	keine Ziffer
\s	Whitespace
\S	kein Whitespace
.	Zeichen
+	voriger Ausdruck mindestens einmal
*	voriger Ausdruck beliebig oft
?	voriger Ausdruck null- oder einmal

Tabelle 8.8 Sonderzeichen bei regulären Ausdrücken

Jedes dieser Metazeichen lässt sich auch wie üblich mit dem Backslash (\) maskieren.

Konstruktor von »Boost.Regex«

Um die regulären Ausdrücke zu verwenden, muss die Header-Datei <*boost/regex.hpp*> mit eingebunden werden (und die Bibliothek hinzugelinkt werden). Ein regulärer Ausdruck mit `regex++` ist ein Typ vom Objekt `basic_regex`. Zunächst soll ein kurzer Überblick über die Klasse `basic_regex` erfolgen, und anschließend werden die drei wichtigsten Algorithmen dieser Bibliothek näher beschrieben. Ein detaillierter Blick in die Klasse `basic_regex` würde zu weit führen, mir geht es eher darum zu demonstrieren, wie man `regex++` in der Praxis einsetzen kann. Details wie weitere Optionen (und davon bietet `regex++` eine Menge) sollte jeder selbst in der Dokumentation nachlesen. Zunächst findet man hier wieder viele Konstruktoren (einschließlich des Kopierkonstruktors und des Destruktors) – von denen man wohl am besten folgenden beschreiben sollte:

```
explicit basic_regex (
    const charT* p,
    flag_type f=regex_constants::normal);
```

Dieser Konstruktor erwartet eine Sequenz von Zeichen (`p`), die den regulären Ausdruck enthalten. Mit dem zweiten Parameter können Sie noch zusätzliche Optionen angeben. Geben Sie hierbei nichts an, wird die Standardeinstellung (`regex_constants::normal`) verwendet. Sofern der reguläre Ausdruck `p` falsch ist, wird entweder eine Exception vom Typ `bad_expression` oder `regex_error` geworfen. Im Grunde bedeuten beide Exceptions dasselbe. Künftig soll `bad_expression` durch `regex_error` ausgetauscht werden.

Neben den Konstruktoren finden Sie außerdem noch einige Methoden und `typdefs`. Im Mittelpunkt des Interesses stehen aber jetzt `regex_match()`, `regex_search()` und `regex_replace()`.

Verwendung von »Boost.Regex«

Wie bereits erwähnt, benötigen Sie die Header-Datei *<boost/regex.hpp>* und die entsprechende Bibliothek, um dieses Beispiel auszuführen. Es kann sein, dass Sie sich diese Bibliothek erst mal besorgen müssen (*www.boost.org*). Da es hierbei leider viele Compiler gibt, müssen Sie sich selbst darum kümmern, wie Sie diese Bibliothek auf Ihrem System/Compiler installieren können. Auch hierzu finden Sie auf der entsprechenden Webseite eine Anleitung.

Zunächst müssen Sie eine Variable vom Typ `basic_regex` deklarieren. Hierbei wird auch gleich der reguläre Ausdruck angegeben. Dazu müssen Sie beim Konstruktor lediglich einen String als Parameter angeben, der den regulären Ausdruck enthält, den Sie verwenden wollen:

```
boost::regex ex("\\d{3}");
```

Mit diesem regulären Ausdruck werden Sie anschließend die Eingabe überprüfen, ob 3 (`{3}`) Dezimalzahlen (`\d` = Digit) eingegeben werden. Dies ist eine häufig verwendete Möglichkeit, um die Eingabe des Anwenders zu überprüfen. Dieser reguläre Ausdruck ist jetzt bereit, um in einen Algorithmus eingesetzt zu werden, beispielsweise mit `regex_match()`:

```
bool b = regex_match(einString, ex);
```

Die Funktion `regex_match()` gibt `true` zurück, wenn der Ausdruck `einString` ein String ist mit exakt 3 Dezimalziffern.

[»] **Hinweis**

Beachten Sie bitte, dass Sie die Escape-Sequenz durch einen vorangestellten Backslash außer Kraft setzen müssen. Statt zum Beispiel `\d` für eine Dezimalzahl müssen Sie `\\d` im String einsetzen, sonst verwendet der Compiler den ersten Backslash als Escape-Zeichen.

Hierzu ein solches Beispiel, mit dem Sie die Eingabe des Anwenders überprüfen können. Folgende reguläre Ausdrücke wollen wir als Beispiel verwenden:

```
// Überprüfen, ob es sich um eine gültige
// E-Mail-Adresse handelt
boost::regex ex(
   "[\\w-]+(?:\\.[\\w-]+)*@(?:[\\w-]+\\.)+[a-zA-Z]{2,7}"
  );

// Überprüfen, ob eine 3-stellige
// Dezimalzahl eingegeben wurde
boost::regex ex("\\d{3}");

// Überprüft, ob nur die Zeichen von
// a-z bzw. A-Z eingegeben wurden.
boost::regex ex("[a-zA-Z]+");
```

Zunächst soll der Algorithmus `regex_match()` in der Praxis verwendet werden:

```
// regex1.cpp
#include <iostream>
#include <boost/regex.hpp>
using namespace std;

bool check_email(const string s) {
   // Gültige E-Mail-Adresse? - trifft 99% aller Adressen
   static const boost::regex ex(
   "[\\w-]+(?:\\.[\\w-]+)*@(?:[\\w-]+\\.)+[a-zA-Z]{2,7}"
   );

   return regex_match(s, ex);
}

bool validate_input(const string s) {
   // 3 Dezimalzahlen eingegeben ...
   static const boost::regex ex("\\d{3}");
   return regex_match(s, ex);
}

bool chars_only(const string s) {
```

```
   // abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
   static const boost::regex ex("[a-zA-Z]+");
   return regex_match(s, ex);
}

int main() {
   string str;
   string str2 = "Suche was in Hallo Welt mit REs";

   cout << "Ihre E-Mail-Adresse bitte: ";
   cin >> str;
   if( check_email( str ) ) {
      cout << "E-Mail-Adresse ist gültig" << endl;
   }
   else {
      cout << "E-Mail-Adresse ist ungültig" << endl;
   }

   cout << "Bitte eine dreistellige Zahl eingeben: ";
   cin >> str;
   if( validate_input ( str ) ) {
      cout << "OK -> die Zahl ist 3-stellig" << endl;
   }
   else {
      cout << "Fehler -> Die Zahl ist nicht 3-stellig"
           << endl;
   }

   cout << "Bitte eine Eingabe machen: ";
   cin >> str;
   if( chars_only ( str ) ) {
      cout << "Die Eingabe enthält nur Zeichen" << endl;
   }
   else {
      cout << "Die Eingabe enthält nicht nur Zeichen"
           << endl;
   }
   return 0;
}
```

Das Programm bei der Ausführung – hier mit G++ unter Linux (Ubuntu):

```
$ g++ -o regex1 regex1.cpp -lboost_regex
$ ./regex1
Ihre E-Mail-Adresse bitte: pronix@pronix.de
E-Mail-Adresse ist gültig
```

```
Bitte eine dreistellige Zahl eingeben: 123
OK -> die Zahl ist 3-stellig
Bitte eine Eingabe machen: EinZweiDrei
Die Eingabe enthält nur Zeichen
$ ./regex1
Ihre E-Mail-Adresse bitte: pronix@pronix
E-Mail-Adresse ist ungültig
Bitte eine dreistellige Zahl eingeben: 1234
Fehler -> Die Zahl ist nicht 3-stellig
Bitte eine Eingabe machen: Eins2Drei
Die Eingabe enthält nicht nur Zeichen
```

[»] **Hinweis**

Der Algorithmus `regex_match()` ist mehrfach überladen und auch in mehreren Versionen vorhanden. Für weitere Details sollten Sie die Dokumentation verwenden.

Suchen

Die Suche mit `regex_search()` unterscheidet sich von `regex_match()` dadurch, dass mit `regex_search()` nicht die kompletten Daten durchsucht werden, sondern nur ein bestimmter Teil davon. Gewöhnlich verwendet man für die Funktion `regex_search()` vier Parameter:

```
regex_search( it1, it2, m, ex )
```

Mit `it1` gibt man die Anfangs- und mit `it2` die Endposition an, an der nach dem regulären Ausdruck `ex` gesucht werden soll. Wurde der Ausdruck gefunden, gibt `regex_search()` `true`, ansonsten `false` zurück. Genauere Informationen können Sie aus dem Parameter `m` ermitteln. Dieser Parameter ist wiederum eine Klasse vom Typ `match_result` und besitzt weitere Methoden. Mit der Methode `matched[n]` lässt sich ermitteln, ob ein bestimmter Ausdruck gefunden wurde. Beispielsweise können Sie mit der Angabe von

```
if( m[1].matched ) {
   // Ausdruck gefunden
}
```

ermitteln, ob der Ausdruck `1` gefunden wurde. Daran lässt sich auch schon erkennen, dass man gleich mehrere Ausdrücke auswerten kann. Auch hierzu empfiehlt es sich, einen Blick in die Dokumentation zu werfen.

[»] **Hinweis**

`regex_search()` bietet außerdem auch noch einen fünften Parameter an, mit dem Sie Flags für weitere Suchoptionen setzen können. Zudem ist der Algorithmus `regex_search()` mehrfach überladen und auch in mehreren Versionen vorhanden.

```
// regex2.cpp
#include <iostream>
#include <fstream>
#include <string>
#include <boost/regex.hpp>
using namespace std;

// Datei zum Durchsuchen laden ...
void load_file(string& s, istream& is) {
   s.erase();
   s.reserve(is.rdbuf()->in_avail());
   char c;
   while(is.get(c)) {
      if(s.capacity() == s.size()) {
         s.reserve(s.capacity() * 3);
      }
      s.append(1, c);
   }
}

int main() {
   string str, file, tmp, re;
   int count = 0;
   boost::smatch m;

   cout << "Welche Datei wollen Sie durchsuchen: ";
   cin >> file;
   ifstream is( file.c_str() );
   if( !is ) {
      cout << "Konnte Datei nicht öffnen" << endl;
      return 1;
   }
   load_file( str, is );

   cout << "Wonach wollen Sie in " << file
        << " suchen (nur ganze Wörter): ";
   cin >> tmp;
   // Hierbei sollten nur ganze Wörter verwendet werden
   re = "(" + tmp + ")";
   // Der reguläre Ausdruck
   boost::regex ex(re);

   // Startposition für die Suche
   string::const_iterator it1 = str.begin();
   // Endposition für die Suche
```

```
   string::const_iterator it2 = str.end();

   while( boost::regex_search( it1, it2, m, ex ) ) {

      // Wurde der reguläre Ausdruck gefunden?
      if( m[1].matched ) {
         ++count;
      }
      // Suchposition erneuern
      it1=m[0].second;
   }
   cout << re << " " << count << "-mal gefunden in "
        << file << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
$ g++ -o regex2 regex2.cpp -lboost_regex
$ ./regex2
Welche Datei wollen Sie durchsuchen: regex2.cpp
Wonach wollen Sie in regex2.cpp suchen (nur ganze Wörter): int
(int) 2-mal gefunden in regex2.cpp
$ ./regex2
Welche Datei wollen Sie durchsuchen: regex2.cpp
Wonach wollen Sie in regex2.cpp suchen (nur ganze Wörter): include
(include) 4-mal gefunden in regex2.cpp
```

Im letzten Abschnitt wurde bereits erwähnt, dass man mit der Funktion `regex_search()` auch mehrere Suchmuster auf einmal verwenden kann. Hierzu dasselbe Beispiel wie *regex2.cpp*, nur jetzt mit mehreren Suchmustern:

```
// regex3.cpp
#include <iostream>
#include <fstream>
#include <string>
#include <boost/regex.hpp>
using namespace std;
void load_file(string& s, istream& is) {
   s.erase();
   s.reserve(is.rdbuf()->in_avail());
   char c;
   while(is.get(c)) {
      if(s.capacity() == s.size()) {
         s.reserve(s.capacity() * 3);
      }
```

```
      s.append(1, c);
   }
}

int main() {
   string str, file, tmp, tmp2, re;
   int count = 0, count2 = 0;
   boost::smatch m;

   cout << "Welche Datei wollen Sie durchsuchen: ";
   cin >> file;
   ifstream is( file.c_str() );
   if( !is ) {
      cout << "Konnte Datei nicht öffnen" << endl;
      return 1;
   }
   load_file( str, is );

   // Suche nach zwei Strings
   cout << "Wonach wollen Sie in " << file
        << " suchen (nur ganze Wörter): ";
   cin >> tmp;
   cout << "Wonach wollen Sie noch suchen : ";
   cin >> tmp2;
   // Hierbei sollten nur ganze Wörter verwendet werden
   re = "(" + tmp + ")" + "|" + "(" + tmp2 + ")";
   // Der reguläre Ausdruck
   boost::regex ex(re);

   // Startposition für die Suche
   string::const_iterator it1 = str.begin();
   // Endposition für die Suche
   string::const_iterator it2 = str.end();

   while( boost::regex_search( it1, it2, m, ex ) ) {
      // Wurde der reguläre Ausdruck gefunden?
      if( m[1].matched ) {
         ++count;
         cout << tmp << " gefunden" << endl;
      }
      else if( m[2].matched ) {
         ++count2;
         cout << tmp2 << " gefunden" << endl;
     }
      // Suchposition erneuern
```

```
      it1=m[0].second;
   }
   cout << tmp << " " << count << "-mal gefunden in "
        << file << endl;
   cout << tmp2 << " " << count2 << "-mal gefunden in "
        << file << endl;
   return 0;
}
```

Das Programm bei der Ausführung:

```
$ g++ -o regex3 regex3.cpp -lboost_regex
$ ./ regex3
Welche Datei wollen Sie durchsuchen: regex3.cpp
Wonach wollen Sie in regex3.cpp suchen (nur ganze Wörter): int
Wonach wollen Sie noch suchen : include
include gefunden
include gefunden
include gefunden
include gefunden
int gefunden
int gefunden
int 2-mal gefunden in regex3.cpp
include 4-mal gefunden in regex3.cpp
```

Ersetzen mit »Boost.Regex«

Das Ersetzen wird mit dem Algorithmus `regex_replace()` durchgeführt. Wie schon die beiden Algorithmen `regex_match()` und `regex_search()` ist die Funktion `regex_replace()` mehrfach überladen und somit auch in verschiedenen Versionen vorhanden (hierzu sei wieder der Blick in die Dokumentation empfohlen). Meistens wird dabei die Version mit sechs Parametern verwendet, wie beispielsweise:

```
regex_replace( output, it1, it2,
               re, ersetzungs_string, flags );
```

Hierbei werden alle regulären Ausdrücke `re` zwischen `it1` und `it2` durch `ersetzungs_string` ersetzt. Die Ausgabe erfolgt auf den Iterator `output` (was eine Datei, ein String oder auch die Standardausgabe sein kann). Zusätzlich können mit `flags` weitere Optionen angegeben werden.

Als Beispiel sei hier ein »C++ nach HTML«-Konverter gegeben, wie Sie diesen ähnlich auch in der Boost-Dokumentation vorfinden.

```
// regex4.cpp
#include <fstream>
```

```
#include <sstream>
#include <string>
#include <iterator>
#include <boost/regex.hpp>
#include <fstream>
#include <iostream>
using namespace std;

extern const char* pre_expression = "(<)";
extern const char* pre_format = "(&lt;)";

const char* expression_text =
// Präprozessor-Direktiven: index 1
   "(^[[:blank:]]*#(?:[^\\\\\\n]|\\\\[^\\n[:punct:] "
   "[:word:]]*[\\n[:punct:][:word:]])*)|"
// Kommentare: index 2
   "(//[^\\n]*|/\\*.*?\\*/)|"
// Literale: index 3
   "\\<([+-]?(?:(?:0x[[:xdigit:]]+)|(?:(?:[[:digit:]]*\\.) "
   "?[[:digit:]]+(?:[eE][+-]?[[:digit:]]+)?))u?(?: "
   " (?:int(?:8|16|32|64))|L)?)\\>|"
// String-Literale: index 4
   "('(?:[^\\\\']|\\\\.)*'|\"(?:[^\\\\\"]|\\\\.)*\")|"
// Schlüsselwörter: index 5
   "\\<(__asm|__cdecl|__declspec|__export|__far16|"
   "__fastcall|__fortran|__import"
   "|__pascal|__rtti|__stdcall|_asm|_cdecl|__except|"
   "_export|_far16|_fastcall"
   "|__finally|_fortran|_import|_pascal|_stdcall|"
   "__thread|__try|asm|auto|bool"
   "|break|case|catch|cdecl|char|class|const|"
   "const_cast|continue|default|delete"
   "|do|double|dynamic_cast|else|enum|explicit|"
   "extern|false|float|for|friend|goto"
   "|if|inline|int|long|mutable|namespace|new|"
   "operator|pascal|private|protected"
   "|public|register|reinterpret_cast|return|short|"
   "signed|sizeof|static|static_cast"
   "|struct|switch|template|this|throw|true|try|"
   "typedef|typeid|typename|union|unsigned"
   "|using|virtual|void|volatile|wchar_t|while)\\>";

const char* format_string =
   "(?1<font color=\"#008040\">$&</font>)"
   "(?2<I><font color=\"#000080\">$&</font></I>)"
   "(?3<font color=\"#0000A0\">$&</font>)"
```

```
   "(?4<font color=\"#0000FF\">$&</font>)"
   "(?5<B>$&</B>)";

const char* header_text =
   "<HTML>\n<HEAD>\n"
   "<TITLE>Autmoatisch generierter HTML-Code</TITLE>\n"
   "<META HTTP-EQUIV=\"Content-Type\" CON-TENT=\"text/html;"
   " charset=iso-8859-15\">\n"
   "</HEAD>\n"
   "<BODY LINK=\"#0000ff\" VLINK=\"#800080\" "
   "BGCOLOR=\"#ffffff\">\n"
   "<P> </P>\n<PRE>";

const char* footer_text = "</PRE>\n</BODY>\n\n";

void load_file(std::string& s, std::istream& is) {
   s.erase();
   s.reserve(is.rdbuf()->in_avail());
   char c;
   while(is.get(c))    {
      if(s.capacity() == s.size())
          s.reserve(s.capacity() * 3);
      s.append(1, c);
   }
}

int main(int argc, const char** argv) {
   boost::regex e1(expression_text), e2(pre_expression);
   string file, in;

   cout <<
       "(CPP2HTML) Welche C++-Datei wollen Sie oeffnen: ";
   cin >> file;
   // Datei zum Einlesen öffnen
   ifstream fs( file.c_str() );
   if( !fs ) {
      cout << "Konnte Datei nicht oeffnen" << endl;
      return 1;
   }
   // Inhalt der Datei in einen String schreiben
   load_file(in,fs);
   // Neuen Dateinamen generieren
   string new_file = file + ".html";
   cout << "Der Dateiname lautet " << new_file << endl;
```

```
   // Neue Datei anlegen und zum Schreiben öffnen
   ofstream os( new_file.c_str() );
   // HTML-Kopfzeile in die Datei schreiben
   os << header_text;
   // Alle öffnenden '<' durch '&lt;' ersetzen
   // Einen temporären String-Stream dazu verwenden
   ostringstream tmpstrstream( ios::out | ios::binary );
   ostream_iterator<char, char> oit(tmpstrstream);
   // Jetzt die Ersetzung von '<' durch '&lt;' durchführen
   boost::regex_replace( oit, in.begin(), in.end(),
                         e2, pre_format,
                         boost::match_default |
                         boost::format_all);
   // Jetzt das Syntax-Highlighting durchführen und
   // die Ausgabe gleich in die neue Datei schreiben ...
   string s(tmpstrstream.str());
   ostream_iterator<char, char> out(os);
   // Action ...
   boost::regex_replace(out, s.begin(), s.end(),
                        e1, format_string,
                        boost::match_default |
                        boost::format_all);
   // Jetzt nur noch die HTML-Fußzeile hinzufügen
   os << footer_text;
   return 0;
}
```

Das Programm bei der Ausführung:

```
$ g++ -o regex4  regex4.cpp -lboost_regex
$ ./regex4
(CPP2HTML) Welche C++-Datei wollen Sie oeffnen: regex4.cpp
Der Dateiname lautet regex4.cpp.html
```

Anschließend können Sie diese Datei (hier: *regex4.cpp.html*) in einem Webbrowser Ihrer Wahl ansehen.

8.7.2 Boost.Iostreams

Wen es schon immer ein wenig gestört hat, dass die Standard-I/O-Streams recht unflexibel waren, der sollte sich die I/O-Streams von Boost ansehen. Eines der Highlights der Bibliothek ist meines Erachtens das umfangreiche Filterkonzept. Damit ist es ohne großen Aufwand möglich, den Datenaustausch, beispielsweise bei einer Netzwerkverbindung, transparent zu verschlüsseln bzw. wieder zu entschlüsseln. Viele fertige Filter wie beispielsweise gzip, bzip2 oder Konvertierungsfilter mit regulären Ausdrücken usw. sind bereits vorhanden. Es ist außer-

dem nicht allzu kompliziert, noch weitere Filter oder gar eigene Quellen einzufügen.

Im Grunde basiert `Boost.Iostreams` auf Datenquellen (Source), einem Datengraben (Sink) und den Filtern für die Ein- und Ausgabe (InputFilter, OutputFilter). Für den lesenden und schreibenden Zugriff auf die Daten werden die Datenquelle bzw. der Datengraben verwendet, und der Filter regelt diesen Datenverkehr nach Wunsch entsprechend.

[»] **Hinweis**

Auf der Webseite *http://www.boost.org/libs/iostreams/doc/index.html* finden Sie neben einer sehr guten Dokumentation auch einige Beispiele, wie solche Filter implementiert werden.

Hierzu soll ein einfaches Beispiel mit `Boost.Iostreams` gezeigt werden, das lesend auf eine gz-komprimierte Datei zugreift und den Inhalt auf dem Bildschirm ausgibt. Dies ist übrigens eine weitere interessante Alternative, um die Transferrate im Netzwerk niedrig zu halten. Statt des kompletten Dokuments wird eine komprimierte Datei über das Netzwerk versendet und erst beim Empfänger dekomprimiert. Hierzu das Listing:

```
// iostream_out.cpp
#include <fstream>
#include <iostream>
#include <boost/iostreams/filtering_streambuf.hpp>
#include <boost/iostreams/copy.hpp>
#include <boost/iostreams/filter/gzip.hpp>

using namespace boost::iostreams;
using namespace std;

void gzip_cout(ifstream &file) {
   filtering_streambuf<input> in;
   in.push(gzip_decompressor());
   in.push(file);
   copy(in, cout);
}

int main(int argc, char *argv[]) {
   ifstream file(argv[1], ios_base::out|ios_base::binary);
   gzip_cout(file);
   return 0;
}
```

Zunächst wird in der Hauptfunktion ein Objekt der Klasse `ifstream` angelegt. Als Eingabefilter wird ein `filtering_streambuf`-Objekt erzeugt. Als Filter für die Datei verwenden wir den GZIP-Kompressor (`gzip_decompressor()`) auf das `ifstream`-Objekt. Die so dekomprimierten Daten kopieren wir nach `cout`.

Das Programm bei der Ausführung:

```
$ g++ -o iostream_out iostream_out.cpp -lboost_iostreams
$ ls -l
-rw-r--r-- 1 595 2009-03-03 12:47 iostream_out.cpp
$ gzip iostream_out.cpp
$ ls -l
-rw-r--r-- 1 336 2009-03-03 12:48 iostream_out.cpp.gz
$ ./iostream_out iostream_out.bak.gz
#include <fstream>
#include <iostream>
#include <boost/iostreams/filtering_streambuf.hpp>
#include <boost/iostreams/copy.hpp>
   ...
   ...
```

8.7.3 Boost.Filesystem

Auch für die eingeschränkte Datei- und gar nicht vorhandene Verzeichnisfunktionalität der Standard-C++-Bibliothek gibt es mit `Boost.Filesystem` eine tolle Lösung. Das ganze Dateisystem wird bei `Boost::Filesystem` mit der Klasse `path` aufgebaut. Auch die Anwendung dieser Klasse wurde recht einfach gehalten. Die unterschiedlichen Pfadangaben der verschiedenen Systeme lassen sich damit ganz leicht umwandeln. Die Angabe der Pfade kann entweder in einer Boost-eigenen (und somit systemunabhängigen) oder systemtypischen Syntax erfolgen. Beispielsweise so:

```
boost::filesystem::path mypath( "files/datei.txt" )
```

Der String als Pfad wird hierbei automatisch in die systemeigene Pfadangabe konvertiert. Um allerdings die native Systemsyntax für den Pfad zu verwenden, muss man beim zweiten Parameter des `path`-Konstruktors `boost::filesystem::native` verwenden.

Der folgende Codeausschnitt ist ein Beispiel, wie man den Inhalt eines Verzeichnisses auslesen kann, was dem Kommando `ls` von Linux/Unix nachempfunden ist:

```
void ls(const boost::filesystem::path &pfad) {
   if(!boost::filesystem::exists(pfad))
```

```
      cerr << "Der Pfad " << pfad.string()
           << " ist nicht vorhanden" << endl;
   else
      cout << pfad.native_file_string() << endl;
   if(boost::filesystem::is_directory(pfad)) {
      boost::filesystem::directory_iterator end;
      for( boost::filesystem::directory_iterator i(pfad);
           i!=end; ++i )
         ls(*i);
   }
}
```

Zunächst wird mit `boost::filesystem::exists()` überprüft, ob der übergebene Pfad existiert. Existiert der Pfad, wird der Pfadname mit der Methode `native_file_string()` plattformüblich (nativ) ausgegeben. Um den weiteren Inhalt der Verzeichnisse auszugeben, überprüfen wir zunächst, ob es sich bei dem Pfad um ein Verzeichnis handelt (`boost::filesystem::is_directory()`). Um die Verzeichnisse jetzt zu durchlaufen, verwenden wir Iteratoren, von denen bei Boost `directory_iterator` implementiert ist.

Dem Iterator `i` übergeben wir jeweils zunächst das erste Element im Verzeichniseintrag. Der zweite Iterator `end` ohne Argumente ist automatisch der End-Iterator. STL-typisch werden die einzelnen Elemente eines Verzeichnisses mit dem Inkrementoperator durchlaufen. Anschließend erfolgt ein weiterer rekursiver Aufruf von `ls()`. Hierbei sollte man aber aufpassen, dass die Rekursion nicht zu tief wird. Für umfangreichere Aufgaben sollte man den rekursiven Aufruf durch die einfache Ausgabe eines Pfades ersetzen. So erhält man den Inhalt des aktuellen gewünschten Verzeichnisses. Der Aufruf der Funktion erfolgt auf diese Weise:

```
if(argc > 1)
   for( int i=1; i<argc; ++I ) {
      ls( boost::filesystem::path(
          argv[i],boost::filesystem::native ) );
   }
   else {
      ls(boost::filesystem::current_path());
  }
}
```

Falls der Anwender keinerlei Angaben des Pfades mit dem ersten Argument der Kommandozeile macht, wird automatisch der Inhalt des aktuellen Pfades ausgegeben.

Hierbei ist es ein leichtes Unterfangen, diese Funktion um eine Suchfunktion nach einer bestimmten Datei zu erweitern:

```
void findFile( const boost::filesystem::path &pfad,
               const std::string& file_name ) {
   if(!boost::filesystem::exists(pfad))
      cerr << "Der Pfad " << pfad.string()
           << " ist nicht vorhanden" << endl;
   if( pfad.native_file_string() == file_name) {
      cout << "!!!Datei gefunden!!!\n";
   }
   if(boost::filesystem::is_directory(pfad)) {
      boost::filesystem::directory_iterator end;
      for( boost::filesystem::directory_iterator i(pfad);
           i!=end;  ++i )
         findFile(*i, file_name);
   }
}
```

Natürlich bietet `Boost.Filesystem` auch Methoden an, um das Dateisystem zu manipulieren. Einige sind in der folgenden Tabelle aufgelistet.

Methode	Beschreibung
`void rename( path const &from_path, path const &to_path);`	Umbenennen von Dateien und Verzeichnissen
`void copy_file( path const &source_file, path const &target_file);`	Kopieren von Dateien und Verzeichnissen
`create_director( const std::string&);`	Erstellen eines Verzeichnisses
`remove_all( const std::string&);`	Löschen eines Verzeichnisses mitsamt Inhalt; entspricht `rmdir`

Tabelle 8.9 Weitere Methoden von »Boost.Filesystem«

Hinweis **[«]**

Die Bibliothek `Boost.Filesystem` bietet noch eine ganze Menge weiterer Funktionen rund um Dateien und Verzeichnisse an. Aber eine Rechte- und Benutzerverwaltung, wie diese beispielsweise bei Linux-Dateisystemen verwendet wird, stellt sie nicht zur Verfügung.

Zum Schluss hierzu nochmals das komplette Listing, das einiges der Funktionalität von `Boost::Filesystem` demonstrieren soll:

```
// myfilesystem.cpp
#include <boost/filesystem/path.hpp>
#include <boost/filesystem/operations.hpp>
#include <boost/filesystem/exception.hpp>
#include <iostream>

using namespace std;

void ls(const boost::filesystem::path &pfad) {
   if(!boost::filesystem::exists(pfad))
      cerr << "Der Pfad " << pfad.string()
           << " ist nicht vorhanden" << endl;
   else
      cout << pfad.native_file_string() << endl;

   if(boost::filesystem::is_directory(pfad)) {
      boost::filesystem::directory_iterator end;
      for( boost::filesystem::directory_iterator i(pfad);
      i!=end; ++i)
         ls(*i);
   }
}

void findFile( const boost::filesystem::path &pfad,
               const std::string& file_name ) {
   if(!boost::filesystem::exists(pfad))
      cerr << "Der Pfad " << pfad.string()
           << " ist nicht vorhanden" << endl;
   if( pfad.native_file_string() == file_name) {
      cout << "!!!Datei gefunden!!!\n";
  }
  if(boost::filesystem::is_directory(pfad)) {
     boost::filesystem::directory_iterator end;
     for( boost::filesystem::directory_iterator i(pfad);
          i!=end; ++i )
        findFile(*i, file_name);
   }
}

void create_Directory(const std::string& dir_name) {
   try {
      boost::filesystem::create_directory(dir_name);
   }
```

```
   catch(std::exception &e) {
       cerr << "Fehler: " << e.what() << endl;
       return;
   }
}

void remove_Directory(const std::string& dir_name) {
   try {
      cout << boost::filesystem::remove_all(dir_name)
           << " Dateien gelöscht" << endl;
   }
   catch(std::exception &e) {
      cerr << "Fehler: " << e.what() << endl;
      return;
   }
}

int main(int argc,char **argv) {
   if(argc > 1)
      for( int i=1; i<argc; ++i ) {
         ls(boost::filesystem::path(
            argv[i],boost::filesystem::native) );
         findFile(argv[1], "myls");
   }
   else {
      ls(boost::filesystem::current_path());
      findFile(boost::filesystem::current_path(), "myls");
  }
}
```

Das Programm bei der Ausführung:

```
$ g++ -o myfilesystem myfilesystem.cpp -lboost_filesystem
$ ./myfilesystem
...
[ aktuelles Verzeichnis wird aufgelistet ]
$ ./myfilesystem /verzeichnisname
...
[ Alles unterhalb von verzeichnisname wird aufgelistet ]
```

Kaum eine Anwendung kommt heute noch ohne eine gewisse Netzwerkfunktionalität aus. Das Erstellen von Anwendungen mit einer integrierten Netzwerkfunktion ist jedoch nicht unbedingt ein portabler Vorgang. Dass es aber trotzdem möglich ist, Klassen zu erstellen, die auf den verschiedensten Plattformen laufen, soll dieses Kapitel zeigen. Häufig sind es nur einige Details, die hierbei beachtet werden müssen.

9 Netzwerkprogrammierung und Cross-Plattform-Entwicklung in C++

Wer beispielsweise mein Buch zur C-Programmierung kennt (»C von A bis Z«, auf der beiliegenden Buch-CD), verfügt bereits über Grundkenntnisse in der Netzwerkprogrammierung. Doch auch ohne Vorkenntnisse können Sie ohne Bedenken mit diesem Kapitel fortfahren, weil auch hier auf diese Grundlagen eingegangen wird. Der absolute Anfänger kann dieses Kapitel somit von Anfang bis Ende durcharbeiten. Dagegen können Leser, die bereits über Kenntnisse in der Netzwerkprogrammierung in C verfügen, die ersten Seiten nur kurz überfliegen (oder auch nochmals lesen) und gleich mit Abschnitt 9.6, »Cross-Plattform-Development«, beginnen. Die zentrale Frage in diesem Kapitel ist, wie man die Socketprogrammierung in Klassen einteilt und das Ganze objektorientiert in echtem C++ programmiert.

Hinweis **[«]**

Vermutlich erscheint Ihnen die Netzwerkprogrammierung nach dem Lesen dieses Kapitels zunächst als recht einfach, aber Sie sollten sich nicht täuschen lassen. Das Kapitel bietet lediglich einen Einstieg in die Welt der Netzwerkprogrammierung. Deshalb sollten Sie sich gegebenenfalls spezielle Literatur anschaffen. Zwar behandeln die meisten dieser Bücher die Netzwerkprogrammierung in C, aber das ist im Grunde kein Problem, da Sie sich sowieso mit diesen Funktionen auseinandersetzen müssen. In diesem Kapitel machen wir auch nichts anderes, als die C-API zur Netzwerkprogrammierung durch eine objektorientierte Perspektive zu erweitern bzw. sie in eine Klasse einzufügen.

Natürlich stellt sich jetzt noch die Frage, auf welchem System das Ganze laufen soll, denn das Thema Netzwerkprogrammierung fällt nicht mehr in den C++-Standard und ist im Grunde vom System abhängig. Aber im Verlauf des Kapitels werden Sie merken, dass die grundlegenden Prinzipien dieselben sind – wie auch

zum Teil die Funktionsnamen. Und darum geht es auch im zweiten Teil dieses Kapitels: um die Cross-Plattform-Entwicklung. Dabei werden Sie erfahren, wie Sie sinnvoll portable Header-Dateien bzw. Bibliotheken erstellen, die auf den verschiedensten Plattformen ausführbar sind (in diesem Fall unter MS Windows und Linux/UNIX) – dies natürlich wiederum anhand von Beispielen aus der Netzwerkprogrammierung.

[»] **Hinweis**

Die Beispiele werden, gegebenenfalls leicht modifiziert, auf noch mehr Systemen ausführbar sein. Aber als Testsysteme standen mir nur MS Windows- und Linux/UNIX-Plattformen zur Verfügung.

9.1 Begriffe zur Netzwerktechnik

Bevor ich Sie in die Netzwerkprogrammierung einführe, möchte ich Ihnen noch einige Begriffe der Netzwerktechnik etwas näherbringen, damit Sie anschließend bei der Programmierung nicht ins Straucheln geraten. Sollten Sie mit diesen Begriffen bereits vertraut sein, können Sie diesen Abschnitt überspringen.

9.1.1 IP-Nummern

Die *IP-Nummer* (*IP = Internet Protocol*) ist mit einer Telefonnummer vergleichbar. Unter dieser Nummer sind Sie im Internet für alle anderen Teilnehmer erreichbar. Daher ist es auch verständlich, dass diese Nummer eindeutig sein muss, so dass keine Adressenkonflikte auftreten können.

Statische und dynamische IP-Nummern

Bei den IP-Nummern unterscheidet man (sofern man von einem Unterschied sprechen kann) zwischen einer *statischen* und einer *dynamischen* IP-Nummer. Rechner, die ständig im Internet sind, um etwa einen bestimmten Service anzubieten, benötigen im Allgemeinen eine statische IP-Nummer. Solchen Rechnern wird einmal eine Nummer zugeteilt, die sich anschließend nicht mehr ändert. Natürlich können auch Sie sich über einen Internet-Provider oder besser gleich über das *Network Information Center (NIC)* eine eigene IP-Nummer zuteilen lassen, beispielsweise um einen eigenen Webserver zu betreiben. Für Privatpersonen bzw. mittelständische Firmen lohnt sich dies allerdings recht selten, da eine feste IP-Nummer auch ihren Preis hat.

Wenn Sie sich ins Internet einwählen, wird Ihnen in der Regel jedes Mal von Ihrem Internet-Service-Provider eine neue dynamische IP-Adresse zugeteilt. Neben dem geringeren Verbrauch von IP-Adressen reduziert sich auch der

Admin-Aufwand erheblich. Dies ist besonders bei größeren lokalen Netzwerken der Fall. Hier muss der Admin nicht jede einzelne statische IP-Adresse ins Netzwerk integrieren, sondern eine Software übernimmt meistens die Verteilung der dynamischen IP-Adressen.

IPv4- und IPv6-Nummern

Zurzeit setzen sich IP-Nummern (*IPv4*) noch aus vier Zahlen (32 Bit) zwischen 0 und 255 zusammen. Dadurch sind mit IPv4 Adressierungen zwischen `0.0.0.0` und `255.255.255.255` möglich – was etwa 4 Milliarden Adressen ergibt. Dies stimmt allerdings nur theoretisch, da diverse Nummern, zum Beispiel mit der Endung 0 und 255, für andere Zwecke vergeben sind. Und 4 Milliarden IP-Adressen sind in der Tat nicht viele, wenn man dies mit der Anzahl der Weltbevölkerung vergleicht. Somit musste eine andere Lösung gesucht werden, die mit *IPv6* auch schon längst gefunden wurde. In IPv6 sind die Adressen 128 Bit lang statt der 32 Bit in IPv4. Damit lassen sich natürlich erheblich mehr IP-Adressen darstellen.

IPv6 ist allerdings noch nicht eingeführt und nach wie vor auf unbestimmte Zeit Zukunftsmusik. Zudem wird man sich auch an eine andere Schreibweise gewöhnen müssen, die einem auf den ersten Blick ein wenig komplexer erscheint. Schreibt man eine IPv6-Adresse vollständig aus, ergeben sich sieben Doppelpunkte getrennt durch Hexadezimalzahlen; beispielsweise `2ffd:345:34b:33:432:e23:a:2`.

Diese Schreibweise kann noch vereinfacht werden durch die Zusammenfassung von Nullen. So kann zum Beispiel eine Gruppe von aufeinanderfolgenden Nullen durch zwei Doppelpunkte (`::`) angegeben werden. Dies ist auch nötig bzw. der Fall, wenn die Kompatibilität zu den IPv4-Adressen gewahrt werden muss, da beispielsweise bei einer IPv4-Adresse, die in IPv6 konvertiert wurde, die ersten sechs Gruppen mit Nullen beziffert sind (zum Beispiel IPv4: `xxx.xxx.xxx.xxx` konvertiert in IPv6: `0:0:0:0:0:0:xxxx:xxxx`). So kann man es sich auch hier mit der Doppelpunktregelung einfacher machen (`0:0:0:0:0:0:xxxx:xxxx`; Gruppe von Nullen durch `::` ersetzen: `::xxxx:xxxx`). Damit man bei der Konvertierung nicht ständig den hexadezimalen Wert der IPv4-Adresse in die IPv6-Adresse umrechnen muss (oder errechnen lässt), ist es auch erlaubt, die letzten beiden Ziffern dezimal (also aus den vier Zahlen wie in IPv4) anzugeben (also `0:0:0:0:0:0:xxx.xxx.xxx.xxx` oder auch `::xxx.xxx.xxx.xxx`).

9.1.2 Portnummern

Bei den Internetprotokollen TCP/IP und UDP/IP sind neben den IP-Nummern, mit denen ein Rechner spezifiziert wird, auch noch sogenannte *Portnummern* (16 Bit lang, 0 bis 65.535) vorhanden. Mit diesen wird ein bestimmter Service (Dienst) auf dem Rechner spezifiziert. Jeder Service hat dabei gewöhnlich eine

eigene Portnummer. Die bekannteste Portnummer dürfte zum Beispiel 80 (HTTP-Dienst) sein; über diesen Port unterhält sich in der Regel der Webserver mit dem Webbrowser. Analog der Portnummer reagiert also auch der Rechner mit einem entsprechenden Dienst und dem damit verknüpften Protokoll.

Bei Firewalls lassen sich Portnummern einsetzen, um bestimmte Nachrichten herauszufiltern. Außerdem lassen sich dabei bestimmte Dienste auf dem Rechner sperren, indem einfach entschieden wird, welche Ports durchgelassen werden und welche nicht. Man kann sich die Ports auch als Türen vorstellen – lässt man andauernd die Tür offen stehen, braucht man sich über ungebetene Gäste nicht zu wundern.

Selbstverständlich sind die Ports teilweise auch standardisiert. Besonders die ersten 1.023 Portnummern werden von IANA (*Internet Assigned Numbers Authority*) kontrolliert und zugewiesen. Unter Linux/UNIX können sie in der Regel nur durch den Superuser verwendet werden. Hier finden sich die Portnummern vieler Standardanwendungen. So unterhält sich FTP über den Port 21, TELNET über Port 23 oder der Internetverkehr (HTTP) über Port 80. Natürlich sind die Portnummern unter 1.024 auch ein beliebtes Ziel für Hacker.

9.1.3 Host- und Domainname

Zwar wissen Sie jetzt, dass im Internet alles über IP-Nummern erreichbar ist, doch wenn Sie sich mit diesen Nummern durchs WWW hangeln müssten, wäre das Internet heute wohl nicht so erfolgreich. Deshalb wird an die Nummern zusätzlich ein Name angehängt, bestehend aus *Host-* und *Domainname* und eventuell einer *Top-Level-Domain (TLD)*: hostname.domainname.tld; also beispielsweise *www.pronix.de*.

Der Domainname ist der Name, mit dem Sie eine Webseite (zum Beispiel *pronix.de*) im Internet ansprechen. Er wird von hinten nach vorn aufgelöst. Somit steht das *de* für die Top-Level-Domain (wie der Name sagt: das, was ganz vorn steht). Die Top-Level-Domain bezeichnet die Herkunft und/oder die Zugehörigkeit (*.com*; *.gov*; *.net* etc.) des Domainnamens, in unserem Beispiel also *de* für Deutschland. Nach der Top-Level-Domain folgt der eigentliche Domainname, der hier *pronix* lautet.

Der Hostname ist die niedrigste Instanz eines Domainnamens, der, wie gesagt, von hinten nach vorn aufgelöst wird (aber der erste Teil in der üblichen Leserichtung ist). Im Allgemeinen lautet im Internet der Hostname *www* (*www.pronix.de*) – allerdings spricht nichts dagegen, diesen auch anders zu benennen. Er wird nämlich nur empfohlen, weil sich viele Anwender an das *www* gewöhnt haben. Genauso gut können Sie aber auch *abc.pronix.de* verwenden.

9.1.4 Nameserver

Einfach ausgedrückt ist ein *Nameserver* ein Rechner, der für die Umsetzung von Rechnernamen in IP-Nummern verantwortlich ist. Im Internet ist dies vergleichbar mit einem Telefonbuch, in dem zu jedem Teilnehmer eine bestimmte Telefonnummer verzeichnet ist. Der Dienst, der Ihnen diese Arbeit im Internet abnimmt, wird als *Domain Name System (DNS)* bezeichnet. Bei kleineren Netzwerken wie dem Intranet werden die lokalen IP-Nummern meistens in Form einer Tabelle in einer Datei hinterlegt.

9.1.5 IP-Protokoll

Das *IP-Protokoll* ist dafür verantwortlich, dass die Datenpakete von einem Sender über mehrere Netze zum Empfänger transportiert werden. Die Übertragung des IP-Protokolls findet paketorientiert und verbindungslos statt. Es wird dabei nicht garantiert, dass die Pakete in der richtigen Reihenfolge oder überhaupt beim Empfänger ankommen. Außerdem wird keine Empfangsbestätigung vom Empfänger an den Sender zurückgegeben. Die maximale Länge eines IP-Pakets ist auf 65.535 Bytes beschränkt. Da das IP-Protokoll auch für das IP-Routing durch ein Netzwerk verantwortlich ist, kann dieses Protokoll die einzelnen Stationen auch anhand der IP-Adresse identifizieren. Da es keine Garantie dafür gibt, dass die Daten ans Ziel gelangen, gibt es im IP-Header des IP-Pakets ein *Time-to-Live-Feld (TTL)*, in dem die Lebensdauer eines Datagramms (hier des Paketes) festgelegt wird. Dies dient dazu, dass die Datenpakete nicht unendlich durch das Netz irren und den Datenverkehr belasten. Ist diese Lebensdauer abgelaufen, wird das Datenpaket verworfen.

9.1.6 TCP und UDP

Aufbauend auf dem IP-Protokoll gibt es zwei wichtige Transportprotokolle, *TCP/IP (TCP = Transmission Control Protocol*) und *UDP/IP (UDP = User Datagram Protocol)*. Beide Protokolle sind auf der Transportebene angesiedelt.

Der Vorteil von TCP ist, dass hier eine zuverlässige Datenübertragung garantiert wird. Dabei wird sichergestellt, dass ein Paket, das nach einer gewissen Zeit nicht beim Empfänger angekommen ist, erneut gesendet wird. Ebenso wird die richtige Reihenfolge der einzelnen Datenpakete, in der diese losgeschickt werden, von TCP garantiert. Zusammengefasst erhalten Sie beim TCP-Protokoll eine Ende-zu-Ende-Kontrolle (*Peer-to-Peer*), ein Verbindungs-Management (nach dem sogenannten *Handshake-Prinzip*), eine Zeitkontrolle, eine Flusskontrolle sowie eine Fehlerbehandlung der Verbindung. Man spricht bei TCP von einem *verbindungsorientierten* Transportprotokoll.

Das UDP-Protokoll hingegen verwendet einen verbindungslosen Datenaustausch zwischen den einzelnen Rechnern. Mit UDP haben Sie in Ihren Anwendungen die direkte Möglichkeit, Datagramme zu senden, allerdings ohne Garantie für die Ablieferung eines Datagramms beim Empfänger. Ebenso wenig ist sichergestellt, dass die Datagramme in der richtigen Reihenfolge ankommen (es ist sogar auch möglich, dass Datagramme mehrfach ankommen). Der Vorteil des geringeren Verwaltungsaufwandes ist natürlich ein höherer Datendurchsatz, der allerdings mit TCP nicht möglich ist. UDP ist zudem recht interessant für Videoübertragungen oder bei Netzwerkspielen. Hierbei ist es nicht so schlimm, wenn das ein oder andere Datenpaket nicht ankommt. Die UDP-Strategie kann man mit dem »Erst schießen, dann fragen«-Motto vergleichen.

[»] **Hinweis**

In diesem Buch wird vorwiegend das gängigere TCP-Protokoll beschrieben, dennoch wird gegebenenfalls auf die Unterschiede zum UDP-Protokoll eingegangen. Allerdings sei darauf hingewiesen, dass die Unterschiede beider Protokolle in ihrer Verwendung nicht allzu groß sind.

9.1.7 Was sind Sockets?

Ein *Socket* ist eine bidirektionale (Vollduplex) Software-Struktur, die zur Netzwerk- oder Interprozess-Kommunikation verwendet wird. Somit ist ein Socket eine Schnittstelle zwischen einem Prozess (Ihrer Anwendung) und einem Transportprotokoll, was meistens das TCP- oder UDP-Protokoll ist.

In den RFC ist ein Socket als ein 5-Tupel aus Ziel- und Quell-IP-Adresse, Ziel- und Quell-Port und dem Netzwerkprotokoll beschrieben. Sockets werden in UDP und TCP verwendet.

[»] **Hinweis**

RFC (*Requests for Comments*) ist eine Reihe von technischen und organisatorischen Dokumenten zum Internet (ursprünglich ARPANET), die 1969 begonnen wurde.

Seit 1983 verwendet BSD die Netzwerk-Sockets in seiner Berkeley Sockets API. Auch Linux, Solaris und viele andere UNIX-Derivate verwenden die BSD-Sockets. Der Zugriff auf ein Socket erfolgt, ähnlich wie der auf Dateien, mit einem Filedeskriptor (ein einfacher Integer) – nur mit dem Unterschied, dass es bei Sockets nicht um Dateien geht, sondern um Kommunikationskanäle.

MS Windows verwendet eine ähnliche API wie die der Berkeley Sockets, die Windows Sockets (kurz Winsock).

9.2 Header-Dateien zur Socketprogrammierung

Zur Erstellung von Netzwerkanwendungen mit Sockets sind auf den Systemen verschiedene Header-Dateien und unter MS Windows auch eine bestimmte Bibliothek nötig.

9.2.1 Linux/UNIX

Die Header-Dateien für Linux/UNIX und die BSD-Varianten lauten:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
```

Hinweis [«]

Beachten Sie, dass für *BSD beim Kompilieren immer die Header-Datei *<sys/types.h>* noch vor *<sys/socket.h>* inkludiert werden muss, sonst gibt es einen Fehler – nur für den Fall, dass ich vergessen sollte, es zu erwähnen, und Sie unter *BSD eine seltsame Fehlermeldung erhalten.

9.2.2 Windows

Unter MS Windows sollte es ausreichen, die folgende Header-Datei zu inkludieren:

```
#include <winsock.h>
```

Eventuell kann es auch nötig sein, die Header-Datei *<windows.h>* einzubinden – sofern Ihr Compiler eine seltsame Warn- bzw. Fehlermeldung ausgibt. Alternativ zu *<winsock.h>* können Sie hier auch das neuere *<winsock2.h>* inkludieren, was allerdings für die Beispiele im Buch nicht unbedingt nötig ist.

Hinweis [«]

Bei den meisten Windows-Anwendungen macht es keinen Unterschied, ob Sie Winsock oder Winsock2 verwenden. Winsock2 enthält neue Funktionen für einige neue Netzwerkprotokolle (wie z. B. Bluetooth). Es ist auf jeden Fall komplett abwärtskompatibel zum Original-Winsock.

Bibliotheken

Unter MS Windows wird außerdem noch die Winsock- bzw. Winsock2-Bibliothek benötigt. Hierfür sind leider bei den verschiedensten Compilern unterschiedliche Bibliotheksnamen angegeben. Mir bekannte Namen unter Microsoft Visual C++ sind beispielsweise `WSOCK32.LIB` (für Winsock) und `WS2_32.LIB` (für

Winsock2); bei einigen anderen Compilern habe ich hierbei `WINSOCK32.LIB` und `WINSOCK2_32.LIB` verwendet.

Bei der kostenlosen Entwicklungsumgebung Code::Blocks, die z. B. den MinGW-Compiler verwendet, lautet der Bibliotheksname `LIBWSOCK32.a` (Winsock) oder `LIBWS2_32.a` (Winsock2). Dabei müssen Sie bei Ihrem neuen Projekt lediglich eine der beiden Bibliotheken dem Linker mitteilen, indem Sie im *lib*-Verzeichnis des entsprechenden Compilers die entsprechende Bibliothek mitteilen. Wie dies mit den Entwicklungsumgebungen Visual C++ Express Edition 2008 und dem Code::Blocks genau funktioniert, können Sie in der Anleitung auf der Buch-CD nachlesen.

[»] **Hinweis**

Sofern Ihr Compiler hier nicht erwähnt wurde, sollten Sie die entsprechende Dokumentation lesen, die bei Ihrem Compiler mitgeliefert wurde. Meistens allerdings reicht es auch aus, einen Blick in das Verzeichnis der Bibliotheken zu werfen (meistens *lib*) und nach einem entsprechenden Namen zu suchen.

[»] **Hinweis**

Wenn Sie beim Übersetzen einen Linkerfehler wie »undefined reference to…« erhalten, bedeutet dies, dass Ihre Linkeroptionen zur entsprechenden Winsock-Bibliothek falsch sind oder (plausibel, aber kommt häufig vor) Sie hierbei gar keine Angaben gemacht haben.

Die Windows-Programmierung und die (C-)Syntax

Zunächst sollen hier noch ein paar Hinweise zur Syntax der Windows-Programmierung gegeben werden. Sofern Sie noch nichts damit zu tun hatten, wird es wohl eine Weile dauern, bis Sie sich an die Windows-typische C-Schreibweise gewöhnt haben. Auf den ersten Blick erscheint alles ein wenig fremd. Beispielsweise wird statt `unsigned char` der Name `BYTE` oder statt `unsigned long` der Name `DWORD` für die Datentypen verwendet. Aber ein Blick in die Win32-Bibliothek zeigt, dass sich dies durch eine einfache Typdefinition ergibt:

```
typedef unsigned char       BYTE;
typedef unsigned long       DWORD;
```

Dasselbe gilt für Strukturen und andere Datentypen. Außerdem wird bei den Variablen-Namen von vielen Programmierern eine besondere Schreibweise, die sogenannte *ungarische Notation*, verwendet. Hierbei fängt jeder Variablen-Name mit einer Vorsilbe (Präfix) an, die etwas über den Variablen-Typ aussagt, gefolgt vom eigentlichen Namen, der mit einem Großbuchstaben beginnt. Ein Beispiel

ist der Name szPauseName. Das Präfix sz steht hier für »string-zero terminated« und bedeutet »Zeichenkette mit dem Stringende-Zeichen«.

Zwar wird in diesem Buch weitgehend auf die Verwendung einer solchen Syntax bzw. die Verwendung von Win32-Datentypen verzichtet, es sollte aber dennoch erwähnt werden, falls Sie nach Quellcodes oder Dokumentationen zu diesem Thema Ausschau halten.

Winsock initialisieren

Damit unter MS Windows ein Prozess überhaupt Sockets verwenden kann, muss dieser vor jedem Aufruf einer Socket-Funktion initialisiert werden. Durch diese Initialisierung kann ein Prozess die WS2_32.DLL bzw. WINSOCK.DLL verwenden. Dies wird mit dem Systemaufruf WSAStartup() erledigt.

```
int WSAStartup (
  WORD wVersionRequested,
  LPWSADATA lpWSAData
);
```

Diese Funktion muss für jede weitere Socket-Funktion aufgerufen werden. Der erste Parameter ist vom Datentyp WORD (unsigned short) mit 2 Bytes Länge. Mit diesem Parameter geben Sie die Versionsnummer von Winsock an, die Sie verwenden wollen. Dabei legen Sie im Low-Order-Byte die Major-Nummer und im High-Order-Byte die Minor-Nummer (Revisionsnummer) fest. Damit Sie sich nicht mit Bit-Verschiebungen und den Byte-Orders auseinandersetzen müssen, verwenden Sie am besten gleich das Win32-Makro MAKEWORD().

```
WORD MAKEWORD (
   BYTE bLow,  // low-order byte of short value
   BYTE bHigh  // high-order byte of short value
);
```

Wollen Sie nun die Version 1.2 von Winsock einsetzen, müssen Sie nur MAKEWORD(1, 2) verwenden. Für die Version 2.0 schreiben Sie einfach MAKEWORD(2, 0).

Mit dem zweiten Parameter von WSAStartup() geben Sie einen Zeiger auf die Struktur (LP)WSADATA (LP = Long Pointer) an. In dieser Struktur finden Sie Informationen zur Winsock-Version. Allerdings werden Sie diese Struktur im Buch nicht mehr benötigen.

Wenn Sie mit der Anwendung fertig sind, sollten Sie am Ende mit der Funktion WSACleanup() die Verbindung mit WS2_32.DLL oder WINSOCK.DLL wieder beenden bzw. freigeben.

```
int  WSACleanup (void);
```

Damit werden diverse Aufräumarbeiten durchgeführt und ein interner Referenzzähler, der auf `WS2_32.DLL` oder `WINSOCK.DLL` verweist, wird dekrementiert.

9.3 Client-Server-Prinzip

Die Begriffe *Client* und *Server* werden von der Außenwelt oft missverstanden. Häufig stellt sich der Laie unter einem Server eine Hardware (einfach einen kompletten PC) vor. Mit dem Begriff *Client* ist dies recht ähnlich. Beides ist eigentlich falsch, denn sowohl der Server als auch der Client sind ein Stückchen Software, die miteinander kommunizieren können. Diese Kommunikation muss dabei nicht zwangsläufig auf verschiedenen PCs stattfinden. Auch wenn dies (in der Netzwerkprogrammierung) meistens der Fall ist, ist es auch möglich, dass sich beides auf demselben Rechner befindet. Das Client-Server-Prinzip wird auch gerne zur Kommunikation oder zum Austausch von Daten zwischen verschiedenen nicht verwandten Prozessen verwendet (Stichwort: *Interprozess-Kommunikation*). Natürlich ist dies auch mit den Sockets (UNIX Domain Sockets) möglich – doch das ist hier nicht Thema.

Der Server ist schlicht eine Software, die einen bestimmten Dienst oder auch Service anbietet. Der oder meistens die Clients sind ebenfalls ein Stückchen Software, das diesen Dienst bzw. Service verwendet. Einfachstes Beispiel ist der Webserver (wie beispielsweise der Apache HTTP Server). Er bietet häufig viele Dienste an wie FTP oder SSH, aber der am meisten verwendete Dienst dürfte wohl das Anbieten von Webseiten (HTTP) sein. Und der Webbrowser ist dabei der Client, der diesen angebotenen Service des Webservers verwendet. Mit diesem können Sie Webseiten auf Ihrem PC ansehen. Dabei ist es im Grunde egal, ob der Server auf einem Linux/UNIX-System (was größtenteils der Fall ist) und der Client auf einem MS Windows-System (was bei Webbrowsern auch größtenteils der Fall ist) ausgeführt wird. Wichtig ist, dass beide dieselbe Sprache (beispielsweise *HTTP = Hypertext Transfer Protocol*) sprechen.

Bedeutend ist für das Client-Server-Prinzip in der Netzwerkprogrammierung, dass eine Netzwerkverbindung zwischen mindestens zwei Endpunkten bestehen muss. Steht die Verbindung, kann die Kommunikation über das Netzwerk mit den verschiedenen Protokollen (TCP/IP oder UDP/IP) beginnen.

9.3.1 Loopback-Interface

Da die meisten Leser unter Ihnen das Client-Server-Beispiel auf dem lokalen Rechner testen werden, müssen auch ein paar Anmerkungen zum Loopback-Interface gemacht werden. Mit dem Loopback-Interface (`127.0.0.1` oder auch

localhost) können Sie die Netzwerkprotokolle auf dem lokalen Rechner zur Kommunikation verwenden, auch wenn kein Netzwerk vorhanden ist. Der Sinn des Loopback-Interfaces liegt darin, dass manche Kommandos ihre Kommunikation auf dem Netzwerkprotokoll aufbauen. Somit würden ohne Loopback-Interface einige Kommandos auf dem lokalen Rechner gar nicht funktionieren – wozu u. a. auch Ihre Anwendungen zählen, sofern Sie diese lokal testen und verwenden wollen.

9.4 Erstellen einer Client-Anwendung

In diesem Abschnitt soll es darum gehen, welche Funktionen grundlegend verwendet werden, um eine Client-Anwendung zu erstellen.

9.4.1 »socket()« – Erzeugen eines Kommunikationsendpunkts

Der erste Schritt einer Kommunikationsverbindung – egal, ob Server oder Client – besteht immer darin, ein Socket vom Betriebssystem anzufordern. Dabei ist noch nicht relevant, wer mit wem kommunizieren will. Das Erzeugen eines Sockets (Kommunikationsendpunkt) können Sie sich wie das Installieren einer Stromsteckdose vorstellen. Ähnlich wie bei den Stromsteckdosen weltweit, bei denen es ja auch unterschiedliche Formen und Spannungen gibt, muss auch beim Anlegen eines Sockets angegeben werden, was hier alles »eingesteckt« werden kann.

Hierzu erst einmal die Syntax der Funktion socket() für Linux/UNIX:

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Und hier die Syntax für MS Windows:

```
#include <winsock.h>
SOCKET socket(int af, int type, int protocol);
```

Auf beiden Systemen haben diese Funktionen eine fast identische Syntax – abgesehen vom Rückgabewert, der unter MS Windows SOCKET lautet. Allerdings ist SOCKET letztendlich nichts anderes als eine Typdefinition von int, und somit könnten Sie in der Praxis hierfür auch int verwenden. Als Rückgabewert erhalten Sie bei beiden Versionen den Socket-Deskriptor.

Bei einem Fehler gibt die Linux/UNIX-Version –1 zurück. Den Fehler können Sie mit dem Fehlercode von errno auswerten (beispielsweise mit perror() oder strerror()).

Unter MS Windows wird bei einem Fehler die Konstante `SOCKET_ERROR` (ebenfalls mit `-1` definiert) zurückgegeben. Hierbei können Sie den Fehlercode mit der Funktion `WSAGetLastError()` ermitteln.

Mit dem ersten Parameter `domain` bzw. `af` geben Sie die Adressfamilie (= Protokollfamilie) an, die Sie verwenden wollen. Eine komplette Liste aller auf Ihrem System unterstützten Protokolle finden Sie in der Header-Datei *<sys/socket.h>*. Trotzdem hier ein Überblick über die gängigeren und häufiger verwendeten Protokolle:

Adressfamilie	Bedeutung
`AF_UNIX`	UNIX-Domain-Sockets, werden gewöhnlich für lokale Interprozess-Kommunikation verwendet
`AF_INET`	Internet-IP-Protokoll Version 4 (IPv4)
`AF_INET6`	Internet-IP-Protokoll Version 6 (IPv6)
`AF_IRDA`	IRDA-Sockets, beispielsweise via Infrarot
`AF_BLUETOOTH`	Bluetooth-Sockets

Tabelle 9.1 Einige gängige Adressfamilien

Mit dem zweiten Parameter der Funktion `socket()` geben Sie den Socket-Typ an. Damit legen Sie die Übertragungsart der Daten fest. Für Sie sind hierbei erst einmal nur die symbolischen Konstanten `SOCK_STREAM` für TCP und `SOCK_DGRAM` für UDP interessant.

Mit dem dritten Parameter können Sie ein Protokoll angeben, das Sie zur Übertragung nutzen wollen. Verwenden Sie hierfür `0`, was meistens der Fall ist, wird das Standardprotokoll eingesetzt, das dem gewählten Socket-Typ (zweiter Parameter) entspricht. Im Fall von `SOCK_STREAM` wird TCP und bei `SOCK_DGRAM` wird UDP verwendet. Weitere mögliche Werte, ohne jetzt genauer darauf einzugehen, wären hierbei `IPPROTO_TCP` (TCP-Protokoll), `IPPROTO_UDP` (UDP-Protokoll), `IPPROTO_ICMP` (ICMP-Protokoll) und `IPPROTO_RAW` (wird bei RAW-Sockets verwendet). Wenn Sie allerdings für den Socket-Typ zum Beispiel `SOCK_STREAM` angegeben haben und das TCP-Protokoll verwenden wollen, müssen Sie nicht extra noch beim dritten Parameter `IPPROTO_TCP` angeben. Mit der Angabe von `0` wird dieses Protokoll standardmäßig verwendet.

Somit sieht das Anfordern eines Sockets folgendermaßen aus:

```
// Erzeuge das Socket - Verbindung über TCP/IP
sock = socket( AF_INET, SOCK_STREAM, 0 );
if (sock < 0) {
    // Fehler beim Erzeugen des Sockets
}
```

Hinweis [«]

Damit hier keine Missverständnisse entstehen: Das Erzeugen eines Sockets muss auch auf der Server-Seite durchgeführt werden – wie sonst, als über Sockets, will sich ein Client mit dem Server unterhalten?

9.4.2 »connect()« – Client stellt Verbindung zum Server her

Nachdem mit den Sockets die Kommunikationsendpunkte erzeugt wurden, kann der Client nun versuchen, eine Verbindung zum Server-Socket herzustellen. Dies wird mit der Funktion `connect()` versucht, die unter Linux/UNIX folgende Syntax hat:

```
#include <sys/types.h>
#include <sys/socket.h>

int connect (
   int socket,
   const struct sockaddr *addr,
   int addrlen
);
```

Und die Syntax unter MS Windows:

```
#include <winsock.h>

int connect (
   SOCKET s,
   const struct sockaddr FAR* addr,
   int addrlen
);
```

Auch hier unterscheidet sich die Syntax nicht erheblich voneinander, und auch die Bedeutungen der einzelnen Parameter sind wieder dieselben. Bei einer erfolgreichen Ausführung geben beide Funktionen `0` zurück, bei einem Fehler `–1` (gleichwertig unter MS Windows mit `SOCKET_ERROR`). Den Fehler können Sie auch hier wieder mit der Fehlervariablen `errno` (unter Linux/UNIX) oder mit der Funktion `WSAGetLastError()` (unter MS Windows) ermitteln.

Als erster Parameter wird der Socket-Deskriptor erwartet, über den Sie die Verbindung herstellen wollen. Dies ist der Rückgabewert, den Sie von der Funktion `socket()` erhalten haben.

Um eine Verbindung zu einem anderen Rechner aufzubauen, werden logischerweise auch Adressinformationen benötigt, mit denen sich der Client verbinden will. Die Adressinformationen über den gewünschten Verbindungspartner tra-

gen Sie im zweiten Parameter der Funktion `connect()` ein. Um sich mit dem Server zu verbinden, benötigen Sie Informationen über die Adressfamilie (Protokollfamilie), die Portnummer und logischerweise die IP-Adresse. Eingetragen werden diese Informationen mit dem zweiten Parameter der Struktur `sockaddr`, die folgendermaßen definiert ist:

```
struct sockaddr {
   sa_family_t sa_family; // Adressfamilie AF_XXX
   char sa_data[14];      // Protokolladresse(IP- und Portnummer)
 };
```

Da diese Struktur allerdings recht umständlich auszufüllen ist, wurde für IP-Anwendungen eine spezielle Struktur hierfür eingeführt, mit der es möglich ist, die IP-Nummer und die Portnummer getrennt einzutragen.

```
struct sockaddr_in {
   sa_family sin_family;        // Adressfamilie AF_XXX
   unsigned short int sin_port; // Portnummer
   struct in_addr sin_addr;     // IP-Adresse
   unsigned char pad[8];        // Auffüll-Bytes für sockaddr
 };
```

Da beide Strukturen im Speicher gleichwertig sind, reicht es aus, eine einfache Typumwandlung bei `connect()` vorzunehmen. Mit dem letzten Parameter (`addrlen`) von `connect()` geben Sie die Länge in Bytes von `sockaddr` mit dem `sizeof-`Operator an.

Ausfüllen von »sockaddr_in«

In der Strukturvariablen `sin_family` geben Sie die Adressfamilie (Protokollfamilie) an, mit der Sie kommunizieren wollen. Im Allgemeinen nimmt man hier dieselbe Familie wie schon beim ersten Parameter der Funktion `socket()`.

In `sin_port` geben Sie die Portnummer an, über die Sie mit dem Server in Kontakt treten wollen. Wichtig ist dabei, dass Sie den Wert im Network Byte Order angeben. Wenn Sie sich beispielsweise mit einem Webserver verbinden wollen, genügt es nicht, als Portnummer hier einfach »80« hinzuschreiben. Es muss auch auf die verschiedenen Architekturen, die es in heterogenen Netzwerken gibt, Rücksicht genommen werden. Denn auf den verschiedenen Architekturen gibt es unterschiedliche Anordnungen der Bytes zum Speichern von Zahlen. So wird bei der Anordnung gewöhnlich zwischen *Big Endian* und *Little Endian* unterschieden. Man spricht dabei auch vom Zahlendreher. Beim Big Endian wird das Byte mit dem höchsten Wert an der niedrigsten Adresse gespeichert, das zweithöchste an der nächsten Adresse und so weiter. Bei der Anordnung von Little Endian ist

dies genau umgekehrt. Dabei wird das Byte mit dem niedrigsten Wert an der niedrigsten Stelle gespeichert, das zweitniedrigste an der nächsten Stelle usw.

Da man sich mit Big Endian (auch als *Network Byte Order* bezeichnet) auf eine einheitliche Datenübertragung geeinigt hat, brauchen Sie sich keine Gedanken über verschiedene Architekturen zu machen.

Um jetzt aus einer lokal verwendeten Byte-Reihenfolge (*Host Byte Order*) eine Network-Byte-Order-Reihenfolge oder umgekehrt zu konvertieren, stehen Ihnen die folgenden vier Funktionen zur Verfügung:

```
#include <netinet/in.h>

// Rückgabe : network-byte-order
// Parameter: host-byte-order
unsigned short int htons(unsigned short int hostshort);

// Rückgabe : network-byte-order
// Parameter: host-byte-order
unsigned long int htonl(unsigned long int hostlong);

// Rückgabe  :  host-byte-order
// Parameter : network-byte-order
unsigned short int ntohs(unsigned short int netshort);

// Rückgabe :  host-byte-order
// Parameter : network-byte-order
unsigned long int ntohl(unsigned long int netlong);
```

Nicht jeder kennt allerdings die Portnummern für den entsprechenden Dienst. Hierbei kann die Funktion `getservbyname()` helfen. Dieser Funktion übergeben Sie den Namen eines Dienstes und das Transportprotokoll als Parameter. Anschließend sucht `getservbyname()` in einer speziellen Datei nach einem Eintrag, der dazu passt, und gibt die Portnummer zurück. Hierfür gibt es eine spezielle Struktur in der Header-Datei *netdb.h*, mit der Sie an die Informationen zu den entsprechenden Diensten kommen.

```
struct servent {
   char *s_name;      // Offizieller Name des Services
   char **s_aliases;  // Alias-Liste
   int  s_port;       // Portnummer zum Servicenamen
   char *s_proto;     // verwendetes Protokoll
};
```

Hier folgt nun eine kurze Beschreibung der einzelnen Strukturvariablen:

- `s_name` – offizieller Servicename
- `s_aliases` – ein String-Array mit eventuellen Alias-Namen zum Service, falls vorhanden. Das letzte Element in der Liste ist `NULL`.
- `s_port` – die Portnummer zum Servicenamen
- `s_proto` – der Name des verwendeten Protokolls

Und hier die Syntax zu `getservbyname()`:

```
#include <netdb.h>
struct servent *getservbyname (
      const char *name, const char *proto );
```

Bei Angabe des Dienstes `name` und des Protokolls `proto` liefert Ihnen diese Funktion bei Erfolg eine Adresse auf die Information in `struct servent`. Bei einem Fehler wird `NULL` zurückgegeben.

Die IP-Adresse geben Sie in der Strukturvariablen `sin_addr` an. Allerdings wird auch hier die Network-Byte-Order-Reihenfolge erwartet. Dabei ist uns allerdings die Funktion `inet_addr()` (oder die etwas sicherere Alternative `inet_aton()`) behilflich. Sie können die IP-Adresse als String angeben und erhalten einen für `sin_addr` benötigten 32-Bit-Wert im Network Byte Order zurück.

Wenn der Client den Dienst eines Servers verwenden will, muss ihm natürlich dessen IP-Adresse bekannt sein. Meistens gibt ein Endanwender aber als Adresse den Rechnernamen statt der IP-Adresse an, da dieser einfacher zu merken ist. Damit also ein Client aus dem Rechnernamen (beispielsweise *www.google.de*) eine IP-Adresse (216.239.59.99) erhält, wird die Funktion `gethostbyname()` verwendet.

```
#include <netdb.h>
struct hostent *gethostbyname( const char *rechnername );
```

Um also aus einem Rechnernamen eine IP-Adresse und weitere Informationen zu ermitteln, steht ein Nameserver zur Verfügung – dieser Rechner ist für die Umsetzung zwischen Rechnernamen und IP-Nummern zuständig. Selbst auf Ihrem Rechner finden Sie solche Einträge der lokalen IP-Nummern in der Datei */etc/hosts* hinterlegt. Im Internet hingegen werden diese Daten in einer eigenen Datenbank gesammelt.

Um solche Informationen zu den einzelnen Rechnern zu erhalten, ist in der Header-Datei *netdb.h* folgende Struktur definiert:

```
struct hostent {
   char *  h_name;
   char ** h_aliases;
   short   h_addrtype;
   short   h_length;
   char ** h_addr_list;
 };
```

Eine kurze Beschreibung der einzelnen Strukturvariablen:

- `h_name` – offizieller Name des Rechners
- `h_aliases` – ein String-Array, in dem sich eventuelle Alias-Namen befinden. Das letzte Element ist immer `NULL`.
- `h_addrtyp` – hier steht der Adresstyp, der gewöhnlich `AF_INET` für IPv4 ist.
- `h_length` – hier befindet sich die Länge der numerischen Adresse.
- `h_addr_list` – hierbei handelt es sich um ein Array von Zeigern auf die Adressen für den entsprechenden Rechner.

Die Funktion `gethostbyname()` gibt bei Erfolg einen Zeiger auf `struct hostent` des gefundenen Rechners zurück, bei einem Fehler `NULL`.

Die letzte Strukturvariable `pad` in der Struktur `sockaddr_in` wird lediglich als Lückenfüller verwendet, um `sockaddr_in` auf die Größe von `sockaddr` aufzufüllen.

Wenn Sie jetzt alle Strukturvariablen der Struktur `sockaddr_in` mit Werten belegt haben, können Sie die Funktion `connect()` aufrufen und bei stehender (erfolgreicher) Verbindung Daten austauschen (Senden und Empfangen).

Hier folgt nochmals ein Codeausschnitt, der zeigt, wie das »Auffüllen« der Struktur `sockaddr_in` und der anschließende Aufruf der Funktion `connect()` vonstatten geht. In dem Beispiel wird versucht, mit einem Webserver (Port 80; HTTP), dessen IP-Adresse Sie als Argumente in der Kommandozeile übergeben haben, zu verbinden.

```
struct sockaddr_in server;
unsigned long addr;
...
// Alternative zu memset() -> bzero()
memset( &server, 0, sizeof (server));

addr = inet_addr( argv[1] );
memcpy( (char *)&server.sin_addr, &addr, sizeof(addr));
server.sin_family = AF_INET;
server.sin_port = htons(80);
```

```
...
// Baue die Verbindung zum Server auf
if (connect(
   sock,(struct sockaddr*)&server, sizeof(server) ) < 0) {
        // Fehler beim Verbindungsaufbau ...
    }
```

[»] **Hinweis**

Wenn Sie UDP statt TCP verwenden, können Sie auf einen Aufruf von `connect()` verzichten. Dann allerdings müssen Sie die entsprechende Adressinformation bei den Funktionen `sendto()` zum Senden und `recvfrom()` zum Empfangen von Daten ergänzen.

9.4.3 Senden und Empfangen von Daten

Nachdem Sie sich erfolgreich mit dem Server verbunden haben, können Sie anfangen, Daten an den Server zu senden bzw. zu empfangen. Hierzu gibt es jeweils für TCP und UDP ein Funktionspaar. Es war ja bereits davon die Rede, dass man mit Sockets ähnlich wie bei Dateien mit Filedeskriptoren arbeiten kann. Unter Linux/UNIX kann der Austausch von Daten über Sockets auch mit den Systemcalls `read()` und `write()` stattfinden. Allerdings ist dies unter MS Windows erst ab den Versionen NT/2000/XP mit den Funktionen `ReadFile()` und `WriteFile()` möglich.

[»] **Hinweis**

Natürlich ist es auch hier so, dass die Funktionen zum Senden und Empfangen nicht nur für die Clients gelten, sondern auch für die Server-Anwendung.

»send()« und »recv()« – TCP

Zum Senden von Daten von einem Socket für Stream-Sockets (TCP-Sockets) wird gewöhnlich die Funktion `send()` verwendet, die unter Linux/UNIX folgende Syntax besitzt:

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t send ( int socketfd, const void *data,
               size_t data_len, unsigned int flags );
```

Und unter MS Windows mit Winsock sieht die Syntax wieder ähnlich aus:

```
#include <winsock.h>
int send ( SOCKET s, const char FAR* data,
           int data_len, int flags );
```

Vergleicht man diese Funktion mit `write()`, können Sie Parallelen feststellen. Mit dem ersten Parameter geben Sie den Socket-Deskriptor an, über den Sie die Daten senden wollen. Im zweiten Parameter wird ein Zeiger auf den Speicherbereich erwartet, in dem sich die Daten befinden. Die Größe des Speicherbereichs geben Sie mit dem dritten Parameter an. Mit dem letzten Parameter können Sie das Verhalten von `send()` noch beeinflussen. Wird hierbei `0` angegeben, verhält sich `send()` wie die Systemfunktion `write()` zum Schreiben. Ansonsten wäre beispielsweise die symbolische Konstante `MSG_OOP` ein häufig verwendeter Wert, mit dem *Out-of-band-Daten* gesendet werden können. Weitere `flags` entnehmen Sie bitte wieder der entsprechenden Dokumentation (beispielsweise Manual-Page), da hierauf nicht näher eingegangen wird.

Im Falle eines Fehlers liefert `send()` `–1` zurück (was gleichwertig zur Konstante `SOCKET_ERROR` unter MS Windows ist). Welcher Fehler aufgetreten ist, lässt sich wieder mit den üblichen betriebssystembedingten Routinen überprüfen (`errno` unter Linux/UNIX und `WSAGetLastError()` unter MS Windows).

Auch wenn kein Fehler auftritt, ist es sehr wichtig, den Rückgabewert zu überprüfen. Denn bei der Netzwerkprogrammierung sind auch gewisse Grenzen (Bandbreite) vorhanden, das heißt, Sie können nicht unendlich viele Daten auf einmal versenden. Mit der Auswertung des Rückgabewertes können/müssen Sie sich selbst darum kümmern, dass der eventuelle Rest, der nicht gesendet werden konnte, ebenfalls noch verschickt wird. Dies erledigen Sie, indem Sie `data_len` mit dem Rückgabewert von `send()` vergleichen. Durch diese Differenz (`data_len` – Rückgabewert) erhalten Sie die noch nicht gesendeten Daten.

Um Daten von einem Stream-Socket zu empfangen (lesen), wird die Funktion `recv()` verwendet. Die Syntax unter Linux/UNIX lautet:

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t recv ( int socketfd, void *data ,
               size_t data_len, unsigned int flags );
```

Und die Syntax unter MS Windows:

```
#include <winsock.h>
int recv ( SOCKET s,  char FAR* data,
           int data_len,  int flags  );
```

Hier lassen sich mit Ausnahme des letzten Parameters erneut Parallelen zur Systemfunktion `read()` ziehen. Der erste Parameter ist wieder der Socket-Deskriptor der Verbindung, gefolgt von einem Zeiger auf einen Puffer, in den die Daten gelegt werden sollen. Die Länge des Puffers geben Sie mit dem dritten Parameter an, und mit den Flags können Sie das Verhalten von `recv()` beeinflussen. Eine

Angabe von `0` bedeutet auch hier, dass sich `recv()` wie die Funktion `read()` verhält. Ansonsten werden auch hierbei gerne die Konstanten `MSG_OOP` (für Out-of-band-Daten, die gelesen werden können) und `MSG_PEEK` verwendet. Mit `MSG_PEEK` können Daten erneut gelesen werden. Zu weiteren möglichen `flags` sollten Sie bei Bedarf die entsprechende Dokumentation lesen (beispielsweise Manual-Page).

Im Falle eines Fehlers gilt dasselbe wie schon bei der Funktion `send()`. Außerdem kann die Funktion `recv()` auch `0` zurückgeben. Dies bedeutet dann, dass der Verbindungspartner seine Verbindung beendet hat. Ansonsten wird auch mit `recv()` die Anzahl der erfolgreich gelesenen Bytes zurückgeliefert.

»sendto()« und »recvfrom()« – UDP

Für die Funktionen zum Senden und Empfangen von Datagrammen (UDP-Sockets) werden vorzugsweise `sendto()` und `recvfrom()` verwendet. Die Syntax unter Linux/UNIX lautet:

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t  recvfrom( int  s,  void  *buf,  size_t  len,
                   int flags, struct sockaddr *from,
                   socklen_t   *fromlen );

ssize_t  sendto( int  s,  const  void  *msg,  size_t  len,
                 int  flags, const struct sockaddr *to,
                 socklen_t tolen );
```

Und die entsprechende ähnliche Syntax unter MS Windows:

```
#include <winsock.h>

int sendto( SOCKET s, const char FAR * buf,  int len,
            int flags, const struct sockaddr FAR * to,
            int tolen );

int recvfrom( SOCKET s, char FAR* buf, int len,
              int flags,  struct sockaddr FAR* from,
              int FAR* fromlen );
```

Die Bedeutung der einzelnen Parameter sowie des Rückgabewertes entsprechen exakt der Bedeutung der TCP-Gegenstücke `send()` und `recv()`. Hinzugekommen sind am Ende zwei weitere Parameter. Wobei Sie mit dem fünften Parameter einen Zeiger auf die Adresse des Zielrechners (bei `sendto()`) bzw. einen Zeiger

auf die Adresse des Absenders (bei `recvfrom()`) übergeben. Die Angaben entsprechen dabei dem Parameter `sockaddr` der Funktion `connect()`. Mit dem letzten Parameter beider Funktionen geben Sie wieder die Größe der Struktur `sockaddr` an.

Sollten Sie bei einer UDP-Verbindung die `connect()`-Funktion verwenden, können Sie auch die Funktionen `send()` und `revc()` einsetzen. In diesem Fall werden die fehlenden Informationen zur Adresse automatisch ergänzt.

9.4.4 »close()« und »closesocket()«

Sobald Sie mit der Datenübertragung fertig sind, sollten Sie den Socket-Deskriptor wieder freigeben bzw. schließen. Unter Linux/UNIX können Sie dazu, wie beim Lesen und/oder Schreiben einer Datei, ein simples `close()` verwenden.

```
#include <unistd.h>
int close(int s);
```

Unter MS Windows hingegen wird die Funktion `closesocket()` verwendet, die letztendlich, abgesehen von einem anderen Funktionsnamen, dieselbe Wirkung wie ein `close()` unter Linux/UNIX erzielt.

```
#include <winsock.h>
int closesocket( SOCKET s);
```

Beide Funktionen erwarten als Parameter den zu schließenden Socket-Deskriptor und geben bei Erfolg `0` zurück und bei einem Fehler `–1` (gleichwertig zu `SOCKET_ERROR` unter MS Windows). Auch hier kann man den Fehler anhand von `errno` (Linux/UNIX) oder der Funktion `WSAGetLastError()` (MS Windows) ermitteln. Ein Aufruf von `close()` bzw. `closesocket()` beendet außerdem eine TCP-Verbindung sofort.

9.5 Erstellen einer Server-Anwendung

Die Erstellung der Server-Anwendung gestaltet sich nicht viel schwieriger als die der Client-Anwendung. Der Datenaustausch erfolgt genauso wie bei der Client-Anwendung via `send()`/`recv()` (TCP) bzw. `sendto()`/`recvfrom()` (UDP). Der Server muss allerdings keine Verbindung herstellen – dies ist die Aufgabe des Clients. Es ist jedoch die Aufgabe des Servers, Verbindungswünsche anzunehmen. Und um dies zu realisieren, müssen Sie den Server in einen Wartezustand versetzen.

9.5.1 »bind()« – Festlegen einer Adresse aus dem Namensraum

Nachdem Sie auch auf der Server-Seite mit der Funktion `socket()` eine »Steckdose« bereitgestellt haben, müssen Sie zunächst die Portnummer der Server-Anwendung festlegen. Sie wissen ja bereits von der Client-Anwendung, dass mit `connect()` auf eine bestimmte IP-Adresse und eine Portnummer des Servers zugegriffen wird. Unter welcher IP-Adresse und Portnummer der Server auf Anfragen der Clients wartet, müssen Sie mit der Funktion `bind()` festlegen. Somit weisen Sie praktisch einem Socket eine Adresse zu – schließlich ist es durchaus gängig, dass eine Server-Anwendung mehrere Sockets verwendet. Dass hierbei meistens die IP-Adresse die gleiche ist, dürfte klar sein, aber es ist durchaus möglich, die Datenübertragung über mehrere Ports zuzulassen. Die Funktion `bind()` wiederum teilt dem Betriebssystem mit, welches Socket es mit einem bestimmten Port verknüpfen soll. Sobald dann ein Datenpaket eingeht, erkennt das Betriebssystem anhand der Portnummer, für welches Socket das Paket ist.

Die Syntax zur Funktion `bind()` bei Linux/UNIX lautet:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind( int s, const struct sockaddr name, int namelen );
```

Und die ähnliche Syntax bei MS Windows:

```
#include <winsock.h>
int bind( SOCKET s, const struct sockaddr FAR* name,
          int namelen  );
```

Als ersten Parameter übergeben Sie wie immer den Socket-Deskriptor, den Sie mit `socket()` angelegt haben. Mit dem zweiten Parameter geben Sie einen Zeiger auf eine Adresse und Portnummer an. Damit teilen Sie dem System mit, welche Datenpakete für welches Socket gedacht sind. Die Struktur `sockaddr` bzw. (einfacher) `sockaddr_in` und deren Mitglieder wurden bereits ausführlich im Abschnitt zur Funktion `connect()` beschrieben. Allerdings sollte hier noch erwähnt werden, dass ein Rechner häufig über verschiedene Rechner (unter mehreren Adressen) und verschiedene Netze (Internet, Intranet, lokales Netzwerk etc.) erreichbar ist bzw. sein muss. Damit ein Server über alle Netze und IP-Adressen eine Verbindung annimmt, setzt man die IP-Adresse auf `INADDR_ANY` (natürlich im Network Byte Order). Ansonsten geben Sie die IP-Adresse wie gewöhnlich mit der Funktion `inet_addr()` an.

Neben der IP-Adresse ist es außerdem möglich, eine beliebige Portnummer zuzulassen. Hierfür müssen Sie lediglich `0` als Portnummer (im Network Byte Order) verwenden. Welchen Port Sie dann erhalten haben, können Sie mit der Funktion

`getsockname()` im Nachhinein abfragen. Mehr zu dieser Funktion können Sie der entsprechenden Dokumentation entnehmen (beispielsweise Manual-Page).

Mit dem letzten Parameter geben Sie wiederum die Länge der Struktur (zweiter Parameter) in Bytes mit `sizeof()` an.

`bind()` liefert im Fall eines Fehlers –1 (gleichwertig mit dem Fehlercode `SOCKET_ERROR` unter MS Windows). Welcher Fehler aufgetreten ist, können Sie wiederum mit `errno` (Linux/UNIX) bzw. `WSAGetLastError()` (MS Windows) in Erfahrung bringen.

Hierzu ein kurzer Codeausschnitt, der zeigt, wie die Zuweisung einer Adresse auf der Server-Seite in der Praxis realisiert wird:

```
struct sockaddr_in server;

memset( &server, 0, sizeof (server));
// IPv4-Adresse
server.sin_family = AF_INET;
// Jede IP-Adresse ist gültig
server.sin_addr.s_addr = htonl( INADDR_ANY );
// Portnummer 1234
server.sin_port = htons( 1234 );

if(bind(sock,(struct sockaddr*)&server, sizeof(server)) < 0)
 {
    //Fehler bei bind()
 }
```

9.5.2 »listen()« – Warteschlange für eingehende Verbindungen einrichten

Im nächsten Schritt müssen Sie eine Warteschlange einrichten, die auf eingehende Verbindungswünsche eines Clients wartet – man spricht auch vom *Horchen* am Socket auf eingehende Verbindungen. Eine solche Warteschlange wird mit der Funktion `listen()` eingerichtet. Dabei wird die Programmausführung des Servers so lange unterbrochen, bis ein Verbindungswunsch eintrifft. Mit `listen()` lassen sich durchaus mehrere Verbindungswünsche »gleichzeitig« einrichten.

Die Syntax dieser Funktion lautet unter Linux/UNIX wie folgt:

```
#include <sys/types.h>
#include <sys/socket.h>
int listen( int s, int backlog );
```

Unter MS Windows hingegen sieht die Syntax folgendermaßen aus:

```
#include <winsock.h>
int listen( SOCKET s, int backlog );
```

Mit dem ersten Parameter geben Sie wie immer den Socket-Deskriptor an und mit dem zweiten Parameter die Länge der Warteschlange. Die Länge der Warteschlange ist die maximale Anzahl von Verbindungsanfragen, die in eine Warteschlange gestellt werden, wenn keine Verbindungen mehr angenommen werden können.

Der Rückgabewert ist bei Erfolg `0` und auch hier bei einem Fehler `–1` (wie unter MS Windows mit `SOCKET_ERROR`). Den Fehlercode selbst können Sie wieder mit `errno` (Linux/UNIX) bzw. `WSAGetLastError()` (MS Windows) auswerten.

In der Praxis sieht die Verwendung von `listen()` wie folgt aus:

```
if( listen( sock, 5 ) == -1 ) {
   // Fehler bei listen()
}
```

9.5.3 »accept()« und die Server-Hauptschleife

Sobald nun ein oder mehrere Clients Verbindung mit dem Server aufnehmen wollen, können Sie sich darauf verlassen, dass die Funktion `accept()` immer die nächste Verbindung aus der Warteschlange holt (die Sie mit `listen()` eingerichtet haben). Hier die Syntax dazu unter Linux/UNIX:

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int s, struct sockaddr *addr, socklen_t addrlen);
```

Und die ähnliche Syntax unter MS Windows:

```
#include <winsock.h>
SOCKET accept( SOCKET s,
               struct sockaddr FAR* addr,
               int FAR* addrlen );
```

An der Syntax unter MS Windows lässt sich gleich erkennen, dass die Funktion `accept()` als Rückgabewert ein neues Socket zurückgibt. Hierbei handelt es sich um das gleiche Socket mit denselben Eigenschaften wie beim ersten Parameter `s`. Über diesen neuen Socket wird anschließend die Datenübertragung der Verbindung abgewickelt. Ein so akzeptiertes Socket kann allerdings nicht mehr für weitere Verbindungen verwendet werden. Das Original-Socket `s` hingegen bleibt weiterhin für weitere Verbindungswünsche offen.

Hinweis [«]

`accept()` ist eine blockierende Funktion – das heißt, sie blockiert den aufrufenden (Server-)Prozess so lange, bis eine Verbindung vorhanden ist. Sofern Sie die Eigenschaften des Socket-Deskriptors auf »nicht blockierend« ändern, gibt `accept()` einen Fehler zurück, wenn beim Aufruf keine Verbindungen vorhanden sind.

In den zweiten Parameter schreibt `accept()` Informationen (IP-Adresse und Port) über den Verbindungspartner in die Struktur `sockaddr` bzw. `sockaddr_in`. Dies ist logischerweise nötig, damit Sie wissen, mit wem Sie es zu tun haben. `addrlen` wiederum ist die Größe der Struktur `sockaddr` bzw. `sockaddr_in` – allerdings wird diesmal ein Zeiger auf die Größe der Adresse erwartet!

Bei einem Fehler wird `-1` zurückgegeben (gleichbedeutend mit `SOCKET_ERROR` unter MS Windows). Die genaue Ursache des Fehlers können Sie wieder mit `errno` (Linux/UNIX) bzw. `WSAGetLastError()` (MS Windows) ermitteln. Bei erfolgreicher Ausführung von `accept()` wird, wie bereits beschrieben, ein neuer Socket-Deskriptor zurückgegeben.

Ein wichtiger Teil der Server-Programmierung ist außerdem die Server-Hauptschleife. In dieser Schleife wird gewöhnlich die Funktion `accept()` aufgerufen, und darin findet auch im Allgemeinen der Datentransfer zwischen Client und Server statt. Hier ein Beispiel für eine solche Server-Hauptschleife:

```
struct sockaddr_in client;
int sock, sock2;
socklen_t len;

for (;;) {
        len = sizeof( client );
        sock2 = accept(sock,(struct sockaddr*)&client,&len);
        if (sock2 < 0) {
            //Fehler bei accept()
        }
        // Hier beginnt der Datenaustausch
    }
```

Hierzu nochmals die bildliche Darstellung aller Socket-Funktionen für eine TCP-Verbindung zwischen dem Server und dem Client (siehe Abbildung 9.1):

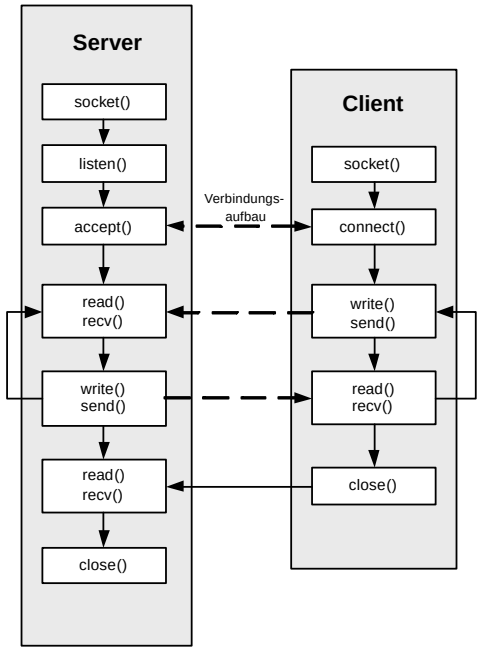

Abbildung 9.1 Kompletter Vorgang einer TCP-Client-Server-Verbindung

Derselbe Vorgang mit allen Socket-Funktionen für eine UDP-Verbindung zwischen Server und Client sieht hingegen so aus (siehe Abbildung 9.2):

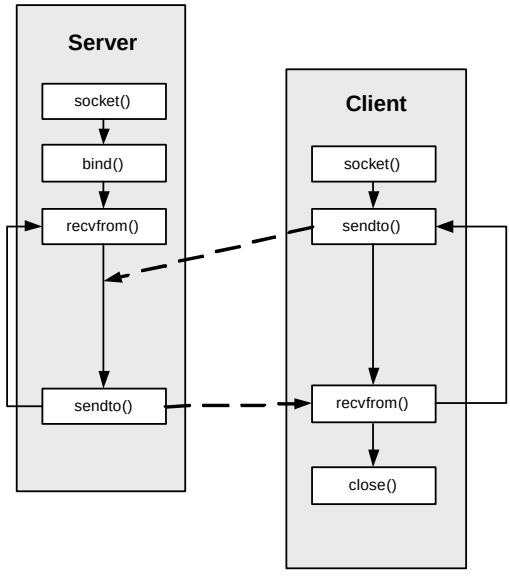

Abbildung 9.2 Kompletter Vorgang einer UDP-Client-Server-Verbindung

9.6 Cross-Plattform-Development

Sie haben jetzt viele Funktionen kennengelernt, mit denen Sie ein schönes C-Programm schreiben könnten, das das Client-Server-Prinzip praktiziert. Aber wie erreicht man Portabilität? Beispielsweise könnten Sie einen Quellcode mit unzähligen `#ifdef`-Direktiven versehen:

```
#ifdef _WIN32        // Haben wir hier MS Windows?
    // Ja, MS Windows ist es, dann nimm den Code hier
#else                // Nein, dann wahrscheinlich Linux/UNIX
    // Vermutlich Linux/UNIX, dann nimm den Code hier
#endif
```

Doch je umfangreicher das Programm wird, desto unsinniger erscheint einem diese Vorgehensweise. Der Code wird hierbei immer komplexer und unübersichtlicher. Bedenken Sie, dass Sie das Ganze auch noch in Klassen verpacken müssen, wir programmieren schließlich in C++. Stellen Sie sich diese Vorgehensweise bei Mammutprojekten (wie zum Beispiel MySQL, Apache etc.) mit mehreren Millionen Zeilen vor, die es auch für viele gängige Plattformen gibt. Das Problem lässt sich dadurch lösen, dass man auf einer abstrakten Ebene (*Abstraction Layer*) programmiert.

9.6.1 Abstraction Layer

Hinter diesem Begriff verbirgt sich nichts Kompliziertes. Der Abstraction Layer isoliert plattformspezifische Funktionen und Datentypen in separate Module für portablen Code. Die plattformspezifischen Module werden dann speziell für jede Plattform geschrieben. Des Weiteren erstellen Sie eine neue Header-Datei, in der sich eventuell die plattformspezifischen `typedef` und `#define` mit den Funktions-Prototypen der Module befinden. Bei der Anwendung selbst binden Sie nur noch diese Header-Datei mit ein. Auf den folgenden Seiten finden Sie nun die einzelnen Quellcodes für unseren abstrakten Layer, den wir später noch erweitern werden.

9.6.2 Header-Datei (»socket.h«)

Nach dem Abschluss der Planung (die wir hier übersprungen haben) sollte man zunächst die Klasse(n) und deren Schnittstelle(n) erstellen – anschließend können Sie sich Gedanken über die Definition der einzelnen Methoden und die Portabilität machen. Hier also zunächst die Header-Datei *socket.h* mit der Klasse `Socket` und den grundlegenden Methoden für MS Windows und Linux/UNIX.

»socket.h« für MS Windows

```
/* socket.h für MS Windows */
#ifndef SOCKET_H_
#define SOCKET_H_
#include <string>
#include <winsock.h>
#include <io.h>
using namespace std;

// Max. Anzahl Verbindungen
const int MAXCONNECTIONS = 5;
// Max. Anzahl an Daten, die auf einmal empfangen werden
const int MAXRECV = 1024;

// Die Klasse Socket
class Socket {
private:
   // Socketnummer (Socket-Deskriptor)
   int m_sock;
   // Struktur sockaddr_in
   sockaddr_in m_addr;

public:
   // Konstruktor
   Socket();
   // virtueller Destruktor
   virtual ~Socket();

   // Socket erstellen - TCP
   bool create();
   // Socket erstellen - UDP
   bool UDP_create();
   bool bind( const int port );
   bool listen() const;
   bool accept( Socket& ) const;
   bool connect ( const string host, const int port );
   // Datenübertragung - TCP
   bool send ( const string ) const;
   int recv ( string& ) const;
   // Datenübertragung - UDP
   bool UDP_send( const string, const string,
                  const int port ) const;
   int UDP_recv( string& ) const;
   // Socket schließen
```

```
    bool close() const;
    // WSAcleanup()
    void cleanup() const;
    bool is_valid() const { return m_sock != -1; }
};
#endif
```

»socket.h« für Linux/UNIX

Die Header-Datei *socket.h* für Linux/UNIX unterscheidet sich im Grunde fast überhaupt nicht von der MS Windows-Version. Es müssen hierbei lediglich die Header-Dateien

```
#include <winsock.h>
#include <io.h>
```

am Anfang der Datei *socket.h* entfernt und durch die Header-Dateien

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <unistd.h>
```

ersetzt werden. Zugegeben, das hätte man auch mit einer `#ifdef`-Direktiven erledigen können, aber ich würde von solchen Direktiven abraten, da es immer wieder Leser gibt, bei denen das dann nicht funktioniert.

Hinweis **[«]**

Wem nicht ganz klar ist, was entfernt und wieder eingefügt werden soll, der findet die Quellcodes auf der Buch-CD.

9.6.3 Quelldatei (»socket.cpp«)

Nachdem Sie die Klasse und deren Schnittstellen nach außen festgelegt haben, können Sie sich die Definitionen dazu vornehmen. Hierbei müssen nun die plattformspezifischen Aspekte beachtet werden. Die Quelldatei *socket.cpp* ist also bereits plattformspezifisch und muss für jede Plattform extra zur Verfügung gestellt werden.

Sicherlich stellt sich zunächst die Frage, wie man die C-Funktionen bei der Definition der Methoden verwendet. In der Klasse `Socket` wurde zum Beispiel eine gleichnamige Methode `bind()` angegeben. Würden Sie in der Definition die C-API-Funktion `bind()` wie folgt in der gleichnamigen Methode `bind()` einsetzen,

```
bool Socket::bind( const int port ) {
   ...
   bind ( m_sock,(struct sockaddr*)&m_addr,sizeof(m_addr) );
   ...
}
```

würden Sie eine Fehlermeldung erhalten. Nicht, weil es diese Funktion nicht gibt, sondern weil versucht wird, die Methode `Socket::bind()` erneut aufzurufen (rekursiver Aufruf). Dass ein Fehler ausgegeben wird, liegt daran, dass die Anzahl der Parameter, mit denen versucht wird, die Methode `Socket::bind()` aufzurufen, nicht mit der Deklaration übereinstimmt.

Wir wollen aber gar nicht die Methode `Socket::bind()`, sondern die C-API-Funktion `bind()` aufrufen. Wenn Sie in Kapitel 3, »Gültigkeitsbereiche, spezielle Deklarationen und Typumwandlungen«, den Abschnitt zu den Gültigkeitsbereichen aufmerksam gelesen haben, wissen Sie noch, dass der Zugriff auf das »globalste« Speicherobjekt über den Scope-Operator möglich ist. Damit also der Zugriff auf die C-API-Funktion `bind()` klappt, müssen Sie nur den Scope-Operator vor die Funktion stellen:

```
bool Socket::bind( const int port ) {
   ...
   ::bind (m_sock,(struct sockaddr*)&m_addr,sizeof(m_addr));
   ...
}
```

»socket.cpp« für MS Windows

```
// socket.cpp
#include <cstdlib>
#include <winsock.h>
#include <io.h>
#include <iostream>
#include "socket.h"
using namespace std;

// Konstruktor
Socket::Socket() : m_sock(0) {
   // Winsock.DLL initialisieren
   WORD wVersionRequested;
   WSADATA wsaData;
   wVersionRequested = MAKEWORD (1, 1);
   if (WSAStartup (wVersionRequested, &wsaData) != 0) {
      cout << "Fehler beim Initialisieren von Winsock"
           << endl;
      exit(1);
```

```
    }
}

// Destruktor
Socket::~Socket() {
    if ( is_valid() )
        ::closesocket ( m_sock );
}

// Erzeugt das Socket - TCP
bool Socket::create() {
    m_sock = ::socket(AF_INET,SOCK_STREAM,0);
    if (m_sock < 0) {
        cout << "Fehler beim Anlegen eines Socket" << endl;
        exit(1);
    }
    return true;
}

// Erzeugt das Socket - UDP
bool Socket::UDP_create() {
    m_sock = ::socket(AF_INET,SOCK_DGRAM,0);
    if (m_sock < 0) {
        cout << "Fehler beim Anlegen eines Socket" << endl;
        exit(1);
    }
}

// Erzeugt die Bindung an die Serveradresse
// - genauer an einen bestimmten Port
bool Socket::bind( const int port ) {
    if ( ! is_valid() ) {
        return false;
    }
    m_addr.sin_family = AF_INET;
    m_addr.sin_addr.s_addr = INADDR_ANY;
    m_addr.sin_port = htons ( port );

    int bind_return = ::bind ( m_sock,
        ( struct sockaddr * ) &m_addr, sizeof ( m_addr ) );
    if ( bind_return == -1 ) {
        return false;
    }
    return true;
}
```

```
// Teile dem Socket mit, dass Verbindungswünsche
// von Clients entgegengenommen werden
bool Socket::listen() const {
   if ( ! is_valid() ) {
      return false;
   }
   int listen_return = ::listen ( m_sock, MAXCONNECTIONS );
   if ( listen_return == -1 ) {
      return false;
   }
  return true;
}

// Bearbeite die Verbindungswünsche von Clients
// Der Aufruf von accept() blockiert so lange,
// bis ein Client Verbindung aufnimmt
bool Socket::accept ( Socket& new_socket ) const {
   int addr_length = sizeof ( m_addr );
   new_socket.m_sock = ::accept( m_sock,
      ( sockaddr * ) &m_addr, ( int * ) &addr_length );
   if ( new_socket.m_sock <= 0 )
      return false;
   else
      return true;
}

// Baut die Verbindung zum Server auf
bool Socket::connect( const string host, const int port ) {
   if ( ! is_valid() )
      return false;
   struct hostent *host_info;
   unsigned long addr;
   memset( &m_addr, 0, sizeof (m_addr));
   if ((addr = inet_addr( host.c_str() )) != INADDR_NONE) {
       /* argv[1] ist eine numerische IP-Adresse */
       memcpy( (char *)&m_addr.sin_addr,
               &addr, sizeof(addr));
   }
   else {
       /* Für den Fall der Fälle: Wandle den Servernamen,  *
        * bspw. "localhost", in eine IP-Adresse um         */
       host_info = gethostbyname( host.c_str() );
       if (NULL == host_info) {
          cout << "Unbekannter Server" << endl;
          exit(1);
```

```
        }
        memcpy( (char *)&m_addr.sin_addr, host_info->h_addr,
                 host_info->h_length);
    }
    m_addr.sin_family = AF_INET;
    m_addr.sin_port = htons( port );

    int status = ::connect ( m_sock,
       ( sockaddr * ) &m_addr, sizeof ( m_addr ) );

  if ( status == 0 )
    return true;
  else
    return false;
}

// Daten versenden via TCP
bool Socket::send( const string s ) const {
    int status = ::send ( m_sock, s.c_str(), s.size(),  0 );
    if ( status == -1 ) {
       return false;
    }
    else {
       return true;
    }
}

// Daten empfangen via TCP
int Socket::recv ( string& s ) const {
  char buf [ MAXRECV + 1 ];
  s = "";
  memset ( buf, 0, MAXRECV + 1 );

  int status = ::recv ( m_sock, buf, MAXRECV, 0 );
  if ( status > 0 || status != SOCKET_ERROR ) {
     s = buf;
     return status;
  }
  else {
     cout << "Fehler in Socket::recv" << endl;
     exit(1);
     return 0;
  }
}
```

```
// Daten versenden via UDP
bool Socket::UDP_
send( const string addr, const string s, const int port ) const {
   struct sockaddr_in addr_sento;
   struct hostent *h;
   int rc;

   h = gethostbyname(addr.c_str());
   if (h == NULL) {
      cout << "Unbekannter Host?" << endl;
      exit(1);
   }
   addr_sento.sin_family = h->h_addrtype;
   memcpy ( (char *) &addr_sento.sin_addr.s_addr,
            h->h_addr_list[0], h->h_length);
   addr_sento.sin_port = htons (port);
   rc = sendto( m_sock, s.c_str(), s.size(), 0,
                (struct sockaddr *) &addr_sento,
                 sizeof (addr_sento));
   if (rc == SOCKET_ERROR) {
      cout << "Konnte Daten nicht senden - sendto()"
           << endl;
      exit(1);
   }
   return true;
}

// Daten empfangen via UDP
int Socket::UDP_recv( string& s ) const {
   struct sockaddr_in addr_recvfrom;
   int len, n;
   char buf [ MAXRECV + 1 ];
   s = "";
   memset ( buf, 0, MAXRECV + 1 );
   len = sizeof (addr_recvfrom);
   n = recvfrom ( m_sock, buf, MAXRECV, 0,
                (struct sockaddr *) &addr_recvfrom, &len );
   if (n == SOCKET_ERROR){
      cout << "Fehler bei recvfrom()" << endl;
      exit(1);
      return 0;
   }
   else {
      s = buf;
      return n;
   }
}
```

```
// Winsock.dll freigeben
void Socket::cleanup() const {
   /* Cleanup Winsock */
   WSACleanup();
}

// Socket schließen und Winsock.dll freigeben
bool Socket::close() const {
   closesocket(m_sock);
   cleanup();
   return true;
}
```

»socket.cpp« für Linux/UNIX

```
// socket.cpp
#include <cstdlib>
#include <iostream>
#include <cstring>
#include "socket.h"
using namespace std;

// Konstruktor
Socket::Socket() : m_sock(0) { }

// Destruktor
Socket::~Socket() {
   if ( is_valid() )
      ::close( m_sock );
}

// Erzeugt das Socket - TCP
bool Socket::create() {
   m_sock = ::socket(AF_INET,SOCK_STREAM,0);
   if (m_sock < 0) {
      cout << "Fehler beim Anlegen eines Socket" << endl;
      exit(1);
   }
   int y=1;
   // Mehr dazu siehe Hinweis am Ende
   setsockopt( m_sock, SOL_SOCKET,
               SO_REUSEADDR, &y, sizeof(int));
   return true;
}
```

```
// Erzeugt das Socket - UDP
bool Socket::UDP_create() {
   m_sock = ::socket(AF_INET,SOCK_DGRAM,0);
   if (m_sock < 0) {
      cout << "Fehler beim Anlegen eines Socket" << endl;
      exit(1);
   }
}

// Erzeugt die Bindung an die Serveradresse
// - genauer an einen bestimmten Port
bool Socket::bind( const int port ) {
   if ( ! is_valid() ) {
      return false;
   }
   m_addr.sin_family = AF_INET;
   m_addr.sin_addr.s_addr = INADDR_ANY;
   m_addr.sin_port = htons ( port );

   int bind_return = ::bind ( m_sock,
      ( struct sockaddr * ) &m_addr, sizeof ( m_addr ) );
   if ( bind_return == -1 ) {
      return false;
   }
   return true;
}

// Teile dem Socket mit, dass Verbindungswünsche
// von Clients entgegengenommen werden
bool Socket::listen() const {
   if ( ! is_valid() ) {
      return false;
   }
   int listen_return = ::listen ( m_sock, MAXCONNECTIONS );
   if ( listen_return == -1 ) {
      return false;
   }
  return true;
}

// Bearbeite die Verbindungswünsche von Clients
// Der Aufruf von accept() blockiert so lange,
// bis ein Client Verbindung aufnimmt
```

```
bool Socket::accept ( Socket& new_socket ) const {
   int addr_length = sizeof ( m_addr );
   new_socket.m_sock = ::accept( m_sock,
     ( sockaddr * ) &m_addr, ( socklen_t * ) &addr_length );
   if ( new_socket.m_sock <= 0 )
      return false;
   else
      return true;
}

// Baut die Verbindung zum Server auf
bool Socket::connect( const string host, const int port ) {
   if ( ! is_valid() )
      return false;
   struct hostent *host_info;
   unsigned long addr;
   memset( &m_addr, 0, sizeof (m_addr));
   if ((addr = inet_addr( host.c_str() )) != INADDR_NONE) {
       /* argv[1] ist eine numerische IP-Adresse */
       memcpy( (char *)&m_addr.sin_addr,
               &addr, sizeof(addr));
   }
   else {
       /* Für den Fall der Fälle: Wandle den Servernamen,  *
        * bspw. "localhost", in eine IP-Adresse um         */
       host_info = gethostbyname( host.c_str() );
       if (NULL == host_info) {
          cout << "Unbekannter Server" << endl;
          exit(1);
       }
       memcpy( (char *)&m_addr.sin_addr, host_info->h_addr,
                host_info->h_length);
   }
   m_addr.sin_family = AF_INET;
   m_addr.sin_port = htons( port );

   int status = ::connect ( m_sock,
      ( sockaddr * ) &m_addr, sizeof ( m_addr ) );

  if ( status == 0 )
    return true;
  else
    return false;
}
```

```
// Daten versenden via TCP
bool Socket::send( const string s ) const {
   int status = ::send ( m_sock, s.c_str(), s.size(),  0 );
   if ( status == -1 ) {
      return false;
   }
   else {
      return true;
   }
}

// Daten empfangen via TCP
int Socket::recv ( string& s ) const {
   char buf [ MAXRECV + 1 ];
   s = "";
   memset ( buf, 0, MAXRECV + 1 );

   int status = ::recv ( m_sock, buf, MAXRECV, 0 );
   if ( status > 0 || status != -1 ) {
      s = buf;
      return status;
   }
   else {
      cout << "Fehler in Socket::recv" << endl;
      exit(1);
      return 0;
   }
}

// Daten versenden via UDP
bool Socket::UDP_
send( const string addr, const string s, const int port ) const {
   struct sockaddr_in addr_sento;
   struct hostent *h;
   int rc;

   h = gethostbyname(addr.c_str());
   if (h == NULL) {
      cout << "Unbekannter Host?" << endl;
      exit(1);
   }
   addr_sento.sin_family = h->h_addrtype;
   memcpy ( (char *) &addr_sento.sin_addr.s_addr,
            h->h_addr_list[0], h->h_length);
   addr_sento.sin_port = htons (port);
```

```
   rc = sendto( m_sock, s.c_str(), s.size(), 0,
                (struct sockaddr *) &addr_sento,
                 sizeof (addr_sento));
   if (rc == -1) {
      cout << "Konnte Daten nicht senden - sendto()"
           << endl;
      exit(1);
   }
   return true;
}

// Daten empfangen via UDP
int Socket::UDP_recv( string& s ) const {
   struct sockaddr_in addr_recvfrom;
   int len, n;
   char buf [ MAXRECV + 1 ];
   s = "";
   memset ( buf, 0, MAXRECV + 1 );
   len = sizeof (addr_recvfrom);
   n = recvfrom ( m_sock, buf, MAXRECV, 0,
                  (struct sockaddr *) &addr_recvfrom,
                  ( socklen_t * )&len );
   if (n == -1){
      cout << "Fehler bei recvfrom()" << endl;
      exit(1);
      return 0;
   }
   else {
      s = buf;
      return n;
   }
}

// Aus Portabilitätsgründen vorhanden
void Socket::cleanup() const { }

// Socket schließen
bool Socket::close() const {
   ::close(m_sock);
   cleanup();
   return true;
}
```

[»] **Hinweis**

In diesem Beispiel zu Linux/UNIX wurde die Funktion `setsockopt()` verwendet. Durch den Einsatz der symbolischen Konstante `SO_REUSEADDR` stellen Sie das Socket so ein, dass sich mehrere Prozesse (Clients) denselben Port teilen können. Das heißt, mehrere Clients können innerhalb kürzester Zeit mit dem Server in Verbindung treten. Außerdem lösen Sie damit auch das Problem, dass der Server beim Neustart seinen lokalen Port erst nach zwei Minuten Wartezeit wieder benutzen kann.

9.6.4 TCP-Echo-Server (Beispiel)

Im folgenden Beispiel wird die Klasse `Socket` in ihrer Anwendung demonstriert. Dazu soll ein einfacher TCP-Echo-Server erstellt werden, der in einer Schleife auf eine Nachricht von einem Client wartet. Neben dem Server müssen Sie also noch einen Client erstellen, der mit der Server-Anwendung in Kontakt tritt und eine Nachricht an diesen sendet. Befindet sich beispielsweise die Server-Software auf einem Rechner mit der IP-Adresse 196.12.32.6, so können Sie mit der Client-Anwendung unter Angabe der entsprechenden IP-Adresse eine einfache Nachricht an den Server senden. Der Server gibt diese Nachricht auf seinem Bildschirm aus.

Um die Klasse `Socket` bei der Server- und Client-Anwendung zu verwenden, müssen Sie nur noch die Header-Datei *socket.h* inkludieren. Vergessen Sie außerdem nicht, beim Übersetzen die Datei *socket.cpp* hinzuzulinken (oder, wenn bereits vorhanden bzw. nur als Objektdatei vorhanden, *socket.o* bzw. *socket.obj*).

TCP-Echo-Server (echo_server.cpp)

```
// echo_server.cpp
#include "socket.h"
#include <string>
#include <iostream>
using namespace std;

int main (void) {
   Socket sock1;
   sock1.create();
   // Wir verwenden Port 15000
   sock1.bind(15000);
   sock1.listen();
   while (true) {
      Socket sock2;
      sock1.accept( sock2 );
      string s;
      sock2.recv (s);
```

```
      cout << "Nachricht von Client erhalten: ";
      cout << s << endl;
      sock2.close();
   }
  sock1.close();
  return 0;
}
```

TCP-Echo-Client (echo_client.cpp)

```
// echo_client.cpp
#include "socket.h"
#include <string>
#include <iostream>
#include <cstdlib>
using namespace std;

int main (int argc, char *argv[]) {
   if( argc < 2 ){
      cout << "Usage: " << *argv << " ServerAdresse\n";
      exit(1);
   }
   Socket sock;
   sock.create();
   // Adresse des Servers
   string argv_1 = argv[1];
   // Mit dem Server auf Port 15000 verbinden
   sock.connect( argv_1, 15000 );
   cout << "Nachricht an den Server: ";
   string s;
   getline(cin, s, '\n' );
   sock.send( s );
   sock.close();
   return 0;
}
```

TCP-Echo-Server – Beispiel ausführen

Hinweis [«]

Bei dem Beispiel wird davon ausgegangen, dass sich die Server-Anwendung `echo_server` auf dem lokalen Rechner unter localhost bzw. 127.0.0.1 befindet. Sollten Sie die Möglichkeit haben, die Server-Anwendung auf einem anderen Rechner auszuführen, und die Client-Anwendung vom lokalen Rechner starten, dann müssen Sie die Client-Anwendung mit der entsprechenden Adresse als zweites Argument starten.

Nachdem Sie die beiden Anwendungen (`echo_server` und `echo_client`) erstellt haben, müssen Sie zunächst den Server starten. Dazu verwenden Sie am besten zwei Konsolen (die MS DOS-Eingabeaufforderung (cmd.exe) unter MS Windows und (Pseudo-)Terminal unter Linux/UNIX).

[»] **Hinweis**

Als *Konsole* wird die Einheit zur Befehlseingabe und zur Ausgabe der Befehlsergebnisse eines Computers genannt, bei Großrechnern auch der Arbeitsplatz des Operators, häufig auch *Terminal* genannt.

Starten Sie also die Server-Anwendung:

```
$> ./echo_server
```

Zunächst passiert gar nichts, da der Server auf die Verbindung und Nachricht einer Client-Anwendung wartet. Starten Sie also einen Client, und geben Sie bei der Eingabeaufforderung etwas ein:

```
$> ./echo_client localhost
Nachricht an den Server: Hallo Server - Ein Test
$>
```

Nachdem Sie bei der Eingabeaufforderung ↵ gedrückt haben, wird die Eingabe an den Server gesendet, und die Client-Anwendung wird sauber beendet. Ein Blick auf das Fenster des Servers sollte Folgendes zeigen:

```
$> ./echo_server
Nachricht von Client erhalten: Hallo Server - Ein Test
```

Selbstverständlich können Sie jetzt eine weitere Client-Anwendung starten und eine Nachricht an den Server senden:

```
$> ./echo_client 127.0.0.1
Nachricht an den Server: Noch eine Nachricht von mir
$>
```

Ein weiterer Blick auf die Konsole, auf der die Server-Anwendung läuft, zeigt jetzt Folgendes:

```
$> ./echo_server
Nachricht von Client erhalten: Hallo Server - Ein Test
Nachricht von Client erhalten: Noch eine Nachricht von mir
```

9.6.5 Exception-Handling integrieren

Mit *socket.h* und *socket.cpp* haben Sie den ersten Schritt für eine echte OOP-Schnittstelle zur Socketprogrammierung getan. Aber im Grunde haben wir bisher

nur eine objektorientierte Hülle über die C-API gestülpt. Die Verwendung der Klasse `Socket` vereinfacht den Umgang mit der C-API sehr, aber es ist noch mehr möglich.

Es fehlt zum Beispiel eine Fehlerbehandlung. Wenn bisher ein Fehler aufgetreten ist, wurde eine Fehlermeldung ausgegeben und das Programm beendet. Dafür sollte man aber für gewöhnlich einen Exception-Handle verwenden. Hierzu habe ich in der Header-Datei *socket.h* eine eigene Klasse `SockExcept` erstellt, die Sie als Grundlage für die Behandlung von Exceptions verwenden können. In unserem Fall macht diese Exception-Klasse nichts anderes, als die Art des Fehlers mit der Methode `get_SockExcept()` auszugeben. Hier die Header-Datei *socket.h* in gekürzter Form:

```
/* socket.h */
#ifndef SOCKET_H_
#define SOCKET_H_
...
using namespace std;
...
// Die Klasse Socket

class Socket {
   // ...
   ...
};

// Exception-Klasse
class SockExcept {
private:
   string except;

public:
   SockExcept( string s ) : except( s ) {};
   ~SockExcept() {};
   string get_SockExcept() { return except; }
};
#endif
```

Jetzt müssen Sie selbstverständlich die Quelldatei *socket.cpp* ändern. Überall, wo eine Fehlermeldung ausgegeben und anschließend das Programm beendet wurde, wird jetzt eine Exception der Klasse `SockExcept` geworfen. Hierzu die veränderte Quelldatei *socket.cpp*:

```
// socket.cpp
#include <cstdlib>
```

```
#include <winsock.h>
#include <io.h>
#include <iostream>
#include "socket.h"
using namespace std;

// Konstruktor
Socket::Socket() : m_sock(0) {
   // Winsock.DLL initialisieren
   WORD wVersionRequested;
   WSADATA wsaData;
   wVersionRequested = MAKEWORD (1, 1);
   if (WSAStartup (wVersionRequested, &wsaData) != 0) {
      throw SockExcept(
             "Fehler beim Initialisieren von Winsock");
   }
}

// Destruktor
Socket::~Socket() {

   if ( is_valid() )
      ::closesocket ( m_sock );
}

// Erzeugt das Socket - TCP
bool Socket::create() {
   m_sock = ::socket(AF_INET,SOCK_STREAM,0);
   if (m_sock < 0) {
      throw SockExcept("Fehler beim Anlegen eines Socket");
   }
   return true;
}

// Erzeugt das Socket - UDP
bool Socket::UDP_create() {
   m_sock = ::socket(AF_INET,SOCK_DGRAM,0);
   if (m_sock < 0) {
      throw SockExcept("Fehler beim Anlegen eines Socket");
   }
}

// Erzeugt die Bindung an die Serveradresse
// - genauer an einen bestimmten Port
bool Socket::bind( const int port ) {
   if ( ! is_valid() ) {
```

```
      return false;
   }
   m_addr.sin_family = AF_INET;
   m_addr.sin_addr.s_addr = INADDR_ANY;
   m_addr.sin_port = htons ( port );

   int bind_return = ::bind ( m_sock,
      ( struct sockaddr * ) &m_addr, sizeof ( m_addr ) );
   if ( bind_return == -1 ) {
      return false;
   }
   return true;
}

// Teile dem Socket mit, dass Verbindungswünsche
// von Clients entgegengenommen werden
bool Socket::listen() const {
   if ( ! is_valid() ) {
      return false;
   }
   int listen_return = ::listen ( m_sock, MAXCONNECTIONS );
   if ( listen_return == -1 ) {
      return false;
   }
  return true;
}

// Bearbeite die Verbindungswünsche von Clients
// Der Aufruf von accept() blockiert so lange,
// bis ein Client Verbindung aufnimmt
bool Socket::accept ( Socket& new_socket ) const {
   int addr_length = sizeof ( m_addr );
   new_socket.m_sock = ::accept( m_sock,
      ( sockaddr * ) &m_addr, ( int * ) &addr_length );
   if ( new_socket.m_sock <= 0 )
      return false;
   else
      return true;
}

// Baut die Verbindung zum Server auf
bool Socket::connect( const string host, const int port ) {
   if ( ! is_valid() )
      return false;
   struct hostent *host_info;
```

```
   unsigned long addr;
   memset( &m_addr, 0, sizeof (m_addr));
   if ((addr = inet_addr( host.c_str() )) != INADDR_NONE) {
       /* argv[1] ist eine numerische IP-Adresse */
       memcpy( (char *)&m_addr.sin_addr, &addr,
                sizeof(addr));
   }
   else {
       /* Für den Fall der Fälle: Wandle den Servernamen, *
        * bspw. "localhost", in eine IP-Adresse um        */
       host_info = gethostbyname( host.c_str() );
       if (NULL == host_info) {
          throw SockExcept("Unbekannter Server");
       }
       memcpy( (char *)&m_addr.sin_addr, host_info->h_addr,
                host_info->h_length);
   }
   m_addr.sin_family = AF_INET;
   m_addr.sin_port = htons( port );

   int status = ::connect ( m_sock,
      ( sockaddr * ) &m_addr, sizeof ( m_addr ) );

  if ( status == 0 )
    return true;
  else
    return false;
}

// Daten versenden via TCP
bool Socket::send( const string s ) const {
   int status = ::send ( m_sock, s.c_str(), s.size(),  0 );
   if ( status == -1 ) {
      return false;
   }
   else {
      return true;
   }
}

// Daten empfangen via TCP
int Socket::recv ( string& s ) const {
   char buf [ MAXRECV + 1 ];
   s = "";
   memset ( buf, 0, MAXRECV + 1 );
```

```
   int status = ::recv ( m_sock, buf, MAXRECV, 0 );
   if ( status > 0 || status != SOCKET_ERROR ) {
      s = buf;
      return status;
   }
   else {
      throw SockExcept("Fehler in Socket::recv");
      return 0;
   }
}

// Daten versenden via UDP
bool Socket::UDP_
send( const string addr, const string s, const int port ) const {
   struct sockaddr_in addr_sento;
   struct hostent *h;
   int rc;

   h = gethostbyname(addr.c_str());
   if (h == NULL) {
      throw SockExcept("Unbekannter Host?");
   }
   addr_sento.sin_family = h->h_addrtype;
   memcpy ( (char *) &addr_sento.sin_addr.s_addr,
            h->h_addr_list[0], h->h_length);
   addr_sento.sin_port = htons (port);
   rc = sendto( m_sock, s.c_str(), s.size(), 0,
                (struct sockaddr *) &addr_sento,
                 sizeof (addr_sento));
   if (rc == SOCKET_ERROR) {
      throw SockExcept(
         "Konnte Daten nicht senden - sendto()");
   }
   return true;
}

// Daten empfangen via UDP
int Socket::UDP_recv( string& s ) const {
   struct sockaddr_in addr_recvfrom;
   int len, n;
   char buf [ MAXRECV + 1 ];
   s = "";
   memset ( buf, 0, MAXRECV + 1 );
   len = sizeof (addr_recvfrom);
   n = recvfrom ( m_sock, buf, MAXRECV, 0,
                (struct sockaddr *) &addr_recvfrom, &len );
```

```
   if (n == SOCKET_ERROR){
      throw SockExcept("Fehler bei recvfrom()");
      return 0;
   }
   else {
      s = buf;
      return n;
   }
}

// Winsock.dll freigeben
void Socket::cleanup() const {
   /* Cleanup Winsock */
   WSACleanup();
}

// Socket schließen und Winsock.dll freigeben
bool Socket::close() const {
   closesocket(m_sock);
   cleanup();
   return true;
}
```

Analog verläuft dieses Verfahren auch bei der Quelldatei *socket.cpp* für Linux/UNIX. Der Einsatz in der Praxis der Exception-Klasse `SockExcept` wird weiter unten gezeigt.

9.6.6 Server- und Client-Sockets erstellen (TCP)

Da die Arbeiten der Server- und der Client-Anwendung zwei unterschiedliche Dinge sind, kann man auch gleich zwei verschiedene Klassen erstellen, die die ganze Arbeit noch mehr vereinfachen. Die grundlegenden Arbeiten der Server-Anwendung sind:

- nach dem Erzeugen eines Sockets (`Socket::create()`) an einen bestimmten Port binden (`Socket::bind()`) und auf Verbindungswünsche von Clients warten (`Socket::listen()`)
- die Verbindungswünsche von Clients bearbeiten, also so lange blockieren, bis ein Client eine Verbindung aufbaut (`Socket::accept()`)
- Daten senden (`Socket::send()`)
- Daten empfangen (`Socket::recv()`)
- Verbindung schließen (`Socket::close()`)

Somit sind auf der Server-Seite nicht mehr als fünf Aktionen notwendig. Daher soll jetzt eine erweiterte Klasse `ServerSock` erstellt werden, die diese Anforderungen erfüllt. Wir verwenden die Klasse `Socket` nach wie vor als Basisklasse. `ServerSock` wird also eine abgeleitete Klasse von `Socket` sein.

Noch einfacher sind die Aufgaben bei der Client-Anwendung. Hierbei sind folgende Schritte notwendig:

- Nach dem Erzeugen eines Sockets (`Socket::create()`) wird versucht, eine Verbindung mit dem Server aufzubauen (`Socket::connect()`).
- Daten senden (`Socket::send()`)
- Daten empfangen (`Socket::recv()`)
- Verbindung schließen (`Socket::close()`)

Der Client benötigt im Grunde also noch weniger Arbeit, um an sein Ziel zu kommen. Auch hierfür werden wir eine von `Socket` abgeleitete Klasse mit dem Namen `ClientSock` erstellen, um wieder auf die von der Klasse `Socket` angebotenen Methoden zuzugreifen.

serversock.h (Server-Socket-Klasse)

```
// serversock.h
#ifndef SERVERSOCK_H
#define SERVERSOCK_H

#include <string>
#include "socket.h"
using namespace std;

class ServerSock : private Socket {
public:
   // Konstruktor
   ServerSock ( int port );
   ServerSock (){};
   // Destruktor
   virtual ~ServerSock();
   // Daten senden
   const ServerSock& operator << ( const string& ) const;
   // Daten empfangen
   const ServerSock& operator >> ( string& ) const;
   // Socket schließen
   void close() const;
   // Die Verbindungswünsche von Clients bearbeiten,
   // also so lange blockieren, bis ein Client eine
```

```
   // Verbindung aufbaut
   void accept ( ServerSock& );
};
#endif
```

Interessant dürfte das Überladen der Operatoren << und >> sein. Damit wird es möglich, über

```
socket >> data;
```

Daten aus einem Socket zu empfangen, und mit

```
socket << data;
```

werden Sie Daten an das Socket senden. Dasselbe wird auch mit dem Socket des Clients möglich sein.

serversock.cpp (Server-Socket-Klasse)

Die Definition der einzelnen Methoden der Klasse `ServerSock` dürfte niemanden mehr überraschen. Hierbei verwenden wir alle benötigten Methoden aus der Klasse `Socket` (weshalb Sie die Header-Datei *socket.h* inkludieren müssen). Jetzt klärt sich auch auf, warum bei den Methoden in der Klasse `Socket` häufig der Rückgabetyp `bool` verwendet wurde. Sobald eine dieser Methoden in `Socket` `false` zurückgibt, wird eine Exception der Klasse `SockExcept` geworfen. Hierzu lautet der Quellcode:

```
// serversock.cpp
#include "serversock.h"
#include "socket.h"

ServerSock::ServerSock ( int port ) {
   Socket::Socket();
   if ( ! Socket::create() ) {
      throw SockExcept (
         "ServerSock: Fehler bei Socket::create()");
   }
   if ( ! Socket::bind ( port ) ) {
      throw SockExcept(
         "ServerSock: Fehler bei Socket::bind()");
   }
   if ( ! Socket::listen() ) {
      throw SockExcept(
         "ServerSock: Fehler bei Socket::listen()");
   }
}
```

```
ServerSock::~ServerSock() { }

const ServerSock&
ServerSock::operator << ( const std::string& s ) const {
   if ( ! Socket::send ( s ) ) {
      throw SockExcept (
         "ServerSock: Fehler bei Socket::send()");
   }
   return *this;
}

const ServerSock& ServerSock::operator >> ( std::string& s )
                                                    const {
   if ( ! Socket::recv ( s ) ) {
      throw SockExcept(
         "ServerSock: Fehler bei Socket::recv()");
   }
   return *this;
}

void ServerSock::accept ( ServerSock& sock ) {
   if ( ! Socket::accept ( sock ) ) {
      throw SockExcept(
         "ServerSock: Fehler bei Socket::accept()");
   }
}

void ServerSock::close() const {
   if( ! Socket::close() ) {
      throw SockExcept(
         "ServerSock: Fehler bei Socket::close()");
   }
}
```

clientsock.h (Client-Socket-Klasse)

```
// clientsock.h
#ifndef CLIENT_SOCK_H
#define CLIENT_SOCK_H
#include <string>
#include "socket.h"
using namespace std;

class ClientSock : private Socket {
public:
```

```
   // Konstruktor
   ClientSock( string host, int port );
   // Destruktor
   virtual ~ClientSock(){};
   // Daten senden
   const ClientSock& operator << ( const string& ) const;
   // Daten empfangen
   const ClientSock& operator >> ( string& ) const;
   // Socket schließen
   void close() const;
};
#endif
```

clientsock.cpp (Client-Socket-Klasse)

Auch bei der Definition der Klasse `ClientSock` finden Sie nichts, was Sie nicht schon in ähnlicher Form von den Methoden der Klasse `ServerSock` kennen.

```
// clientsock.cpp
#include "clientsock.h"
#include "socket.h"
using namespace std;

ClientSock::ClientSock( string host, int port ) {
   if ( ! Socket::create() ) {
      throw SockExcept (
         "ClientSock: Fehler bei Socket::create()");
   }
   if ( ! Socket::connect ( host, port ) ) {
      throw SockExcept(
         "ClientSock: Fehler bei Socket::connect()");
   }
}

const ClientSock& ClientSock::operator << (const string& s)
                                                    const {
   if ( ! Socket::send ( s ) ) {
      throw SockExcept(
         "ClientSock: Fehler bei Socket::send()");
   }
   return *this;
}

const ClientSock& ClientSock::operator >> ( string& s )
                                                  const {
```

```
   if ( ! Socket::recv ( s ) ) {
      throw SockExcept(
         "ClientSock: Fehler bei Socket::recv()");
   }
   return *this;
}

void ClientSock::close() const {
   if( ! Socket::close() ) {
      throw SockExcept(
         "ClientSock: Fehler bei Socket::close()");
   }
}
```

All together – die Hauptprogramme (Server/Client)

Zur Demonstration soll nochmals derselbe TCP-Echo-Server verwendet werden. Nur werden Sie jetzt für die Server-Anwendung die Klasse `ServerSock` und für die Client-Anwendung die Klasse `ClientSock` einsetzen. Die Ausführung und Anwendung des Programms bleibt im Grunde dieselbe, nur erhält der Client jetzt nach dem Senden einer Nachricht auch noch eine Bestätigung vom Server, dass er diese Nachricht erhalten hat. Des Weiteren wurde jetzt im Hauptprogramm das Behandeln einer Exception mit der erstellten Exception-Klasse `SockExcept` hinzugefügt:

echo_server2.cpp

```
// echo_server2.cpp
#include "serversock.h"
#include <iostream>
using namespace std;

int main() {
   try {
      ServerSock server ( 15000 );
      while ( true ) {
         ServerSock new_sock;
         server.accept ( new_sock );
         try {
            while ( true ) {
               string data;
               new_sock >> data;
               cout << "Nachricht erhalten: "
                    << data << endl;
               new_sock << "Server hat Daten erhalten";
```

```
                break;
             }
          }
          catch ( SockExcept& ) {}
          new_sock.close();
       }
   }
   catch (SockExcept& e) {
      cout << "Exception wurde ausgeworfen: "
           << e.get_SockExcept() << endl;
   }
   return 0;
}
```

Ein besonderes Augenmerk soll auf das Abfangen der inneren Exception gerichtet werden. Da `Socket::recv()` eine Exception auswirft, wenn keine Daten mehr zum Lesen vorhanden sind, muss dies nicht zwangsläufig das Ende des Programms bzw. einen Fehler bedeuten. Mit dem leeren Exception-Handle ignorieren wir diese Exception. Man könnte gegebenenfalls weitere Maßnahmen treffen, worauf hier aber verzichtet wird.

echo_client2.cpp

```
// echo_client2.cpp
#include "clientsock.h"
#include <string>
#include <iostream>
#include <cstdlib>
using namespace std;

int main (int argc, char *argv[]) {
   if( argc < 2 ){
      cout << "Usage: " << *argv << " ServerAdresse\n";
      exit(1);
   }
   string argv_1 = argv[1];
   try {
      ClientSock client_socket ( argv_1, 15000 );
      cout << "Nachricht an den Server: ";
      string s;
      getline(cin, s, '\n' );
      try {
         client_socket << s;
         client_socket >> s;
         cout << s <<  endl;
```

```
      }
      catch ( SockExcept& ) {}
      client_socket.close();
   }
   catch ( SockExcept& e ) {
      cout << "Eine Exception wurde abgefangen: "
           << e.get_SockExcept() << endl;
   }
   return 0;
}
```

Beispiel ausführen

Hinweis [«]

Wenn Sie das Beispiel übersetzen wollen, beachten Sie bitte, dass Sie auch alle Dateien übersetzen (bzw. zum Projekt hinzugefügt haben). Dies wären bei der Server-Anwendung neben dem Hauptprogramm `echo_server2`: *socket.cpp*, *socket.h*, *serversock.cpp* und *serversock.h*. Auf der Seite des Clients wären dies neben dem Hauptprogramm `echo_client2`: *socket.cpp*, *socket.h*, *clientsock.cpp* und *clientsock.h*.

Hinweis [«]

Auch hier wird davon ausgegangen, dass sich die Server-Anwendung auf dem lokalen Rechner befindet (also localhost bzw. 127.0.0.1). Passen Sie beim Starten der Client-Anwendung gegebenenfalls die Adresse des Servers an (erstes Argument).

Starten Sie auch hier zunächst wieder die Server-Anwendung:

```
$> ./echo_server2
```

Auch hier passiert zunächst gar nichts, und der Server wartet erneut auf einen Client. Starten Sie jetzt einen Client, und kommen Sie dann der Eingabeaufforderung nach:

```
$> ./echo_client2 localhost
Nachricht an den Server: Hallo Server - Jetzt mit ClientSock
Server hat Daten erhalten
$>
```

Prompt bekommen Sie die Meldung, dass der Server die Daten erhalten hat. Ein Blick auf die Konsole der Server-Anwendung sollte dies bestätigen:

```
$> ./echo_server2
Nachricht erhalten: Hallo Server - Jetzt mit ClientSock
```

Auch hier wollen wir einen zweiten Durchgang beginnen und nochmals einen Client starten, um den Server mit Nachrichten zu füllen:

```
$> ./echo_client2 127.0.0.1
Nachricht an den Server: Noch eine Nachricht von mir
Server hat Daten erhalten
$>
```

Beim Server sollte sich nichts Überraschendes mehr ereignen:

```
$> ./echo_server2
Nachricht erhalten: Hallo Server - Jetzt mit ClientSock Nachricht
von Client erhalten: Noch eine Nachricht von mir
```

Einfaches Client-Beispiel – eine komplette Webseite abholen

Wenn Sie der Meinung sind, unser Beispiel eigne sich nur für eigene Server-Client-Beispiele, dann täuschen Sie sich. Im Folgenden finden Sie ein einfaches Client-Beispiel, das eine komplette Webseite, die Sie als erstes Argument in der Kommandozeile angegeben haben, von einem Webserver abholt. Die Daten, die wir an den Webserver senden, bezeichnet man als *HTTP-Request* (*Browser Request*) – eine typische Client-Anfrage, um eine Webseite anzufordern, wie dies jeder Webbrowser macht.

Die Antwort des Servers wird dann als *Server-Response* bezeichnet. Je nach Status der Antwort (`200` bedeutet zum Beispiel, dass alles in Ordnung war), erhalten Sie den Inhalt der angeforderten Webseite.

[»] **Hinweis**

Mehr darüber finden Sie auf der Buch-CD in meinem Buch »C von A bis Z« in Kapitel 26, »CGI mit C«, das sich ausführlicher mit diesem Thema beschäftigt.

[»] **Hinweis**

Wenn Sie das Beispiel übersetzen wollen, beachten Sie bitte, dass Sie auch alle Dateien übersetzen (bzw. zum Projekt hinzugefügt haben). Dies wären neben dem Hauptprogramm *socket.cpp*, *socket.h*, *clientsock.cpp* und *clientsock.h.*

```
// get_website.cpp
#include "clientsock.h"
#include <string>
#include <iostream>
#include <cstdlib>
using namespace std;

int main (int argc, char *argv[]) {
   if( argc < 2 ){
      cout << "Usage: " << *argv << " URL\n";
```

```
      exit(1);
   }
   string argv_1 = argv[1];
   try {
      ClientSock client_socket ( argv_1, 80 );
      client_socket << "GET / HTTP/1.1\n";
      client_socket << "Host: " << argv_1 << "\n";
      client_socket << "User-Agent: Internet Exploiter\n";
      client_socket << "\n\n";
      try {
         while( true ) {
            string s;
            client_socket >> s;
            cout << s << endl;
            cout.flush();
         }
      }
      catch( SockExcept& ) {};
      client_socket.close();
   }
   catch ( SockExcept& e ) {
      cout << "Eine Exception wurde abgefangen: "
           << e.get_SockExcept() << endl;
   }
   return 0;
}
```

Das Programm bei der Ausführung:

```
$> ./get_website www.google.de
HTTP/1.1 200 OK
Cache-Control: private
Content-Type: text/html
9ME; expires=Sun, 17-Jan-2038 19:14:07 GMT; path=/
; domain=.google.de
Server: GWS/2.1
Transfer-Encoding: chunked
Date: Wed, 01 Mar 2006 08:18:21 GMT

b2d
<html>
...
   Hier steht der ganze HTML-Code
...
< html>
```

Wenn Sie nicht wollen, dass der ganze HTML-Code an Ihnen vorbeirauscht, können Sie diesen auch in eine Datei umleiten und anschließend mit einem Webbrowser betrachten:

```
$> ./get_website www.google.de > testfile.html
```

Jetzt finden Sie im aktuellen Verzeichnis, in dem Sie das Programm gestartet haben, eine Datei *testfile.html*.

9.6.7 Ein UDP-Beispiel

Bei unserem Layer haben wir Methoden zum Datenaustausch via UDP geschrieben. Außerdem wurde auch einiges zu UDP erwähnt, so dass ich Ihnen hier ein Client-Server-Beispiel geben möchte.

Allerdings will ich dazu die Header-Datei *socket.h* und die Quelldatei *socket.cpp* verwenden. In der Praxis würde es sich aber auch lohnen, eine eigene Klasse wie `UDP_ServerSock` und `UDP_ClientSock` zu schreiben. Im Gegensatz zu TCP fallen beim UDP-Server die Methoden `Socket::listen()` und `Socket::accept()` weg (siehe auch den Abschnitt zu den UDP-Funktionen). Der Client hingegen kann auf die Methoden `Socket::connect()` verzichten.

Auch hier wollen wir nochmals den Echo-Server verwenden, jedoch basierend auf dem UDP-Protokoll. Wird der Server gestartet, wartet dieser auf die Daten an Port 15.000 eines beliebigen Clients. Der Client wird mit der Adresse des Servers (hier wird wieder von `localhost` bzw. 127.0.0.1 ausgegangen) und einem String als drittem Argument aufgerufen, der an den Server gesendet wird.

UDP-Server – udp_echo_server.cpp

```
// udp_echo_server.cpp
#include "socket.h"
#include <iostream>
using namespace std;

int main() {
   try {
      Socket server;
      server.UDP_create();
      server.bind( 15000 );
      cout << "Server wartet auf Daten ..." << endl;
      try {
         while ( true ) {
            string s;
            server.UDP_recv(s);
            cout << "Daten erhalten: " << s << endl;
```

```
         }
      }
      catch ( SockExcept& ) { }
   }
   catch (SockExcept& e) {
      cout << "Exception wurde ausgeworfen: "
           << e.get_SockExcept() << endl;
   }
   return 0;
}
```

UDP-Client – udp_echo_client.cpp

```
#include "socket.h"
#include <string>
#include <iostream>
#include <cstdlib>
using namespace std;

int main (int argc, char *argv[]) {
   if (argc < 3) {
      cout << "Usage: " << argv[0]
           << " <server> <string>" << endl;
      exit (1);
   }
   try {
      Socket client;
      client.UDP_create();
      client.bind( 15000 );
      string argv_1 = argv[1];
      string argv_2 = argv[2];
      client.UDP_send( argv_1, argv_2, 15000 );
   }
   catch ( SockExcept& e ) {
      cout << "Eine Exception wurde abgefangen: "
           << e.get_SockExcept() << endl;
   }
   return 0;
}
```

Beispiel ausführen

Das Beispiel wird genauso ausgeführt wie schon die Echo-Beispiele der Abschnitte zuvor. Nach dem Starten des UDP-Echo-Servers mit

```
$> ./udp_echo_server
Server wartet auf Daten ...
```

wartet der Server wieder auf die Daten eines Clients. Starten wir also einen Client folgendermaßen:

```
$> ./udp_echo_client localhost Hallo
$>
```

Ein Blick auf den Server sollte jetzt Folgendes zeigen:

```
$> ./udp_echo_server
Server wartet auf Daten ...
Daten erhalten: Hallo
```

Ein zweiter Durchgang soll auch hier gestartet werden:

```
$> ./udp_echo_client 127.0.0.1 Welt
$>
```

Beim Server währenddessen:

```
$> ./udp_echo_server
Server wartet auf Daten ...
Daten erhalten: Hallo
Daten erhalten: Welt
```

9.7 Mehrere Clients gleichzeitig behandeln

Alle Server, die Sie bisher geschrieben haben, konnten immer nur eine Client-Anfrage auf einmal abarbeiten. Alle anderen Clients, die in dieser Zeit mit dem Server in Kontakt traten, mussten in einer Warteschlange warten, bis der Server mit dem Client fertig war und wieder auf Verbindungswünsche (`Socket::accept()`) wartete. Für Anwendungen wie Webserver, Chat-Programme, Spiele-Server usw. ist dieser Umstand ungeeignet.

Wenn Sie jetzt mehrere Clients gleichzeitig behandeln müssen, gibt es abhängig von der Plattform einige sinnvolle Varianten:

- Die Verwendung von (Multi)Threads. Dabei wird für jeden Client ein neuer Thread gestartet. Der Nachteil ist, dass Threads nur bedingt portabel sind, da es auf den verschiedenen Plattformen die unterschiedlichsten Thread-Bibliotheken gibt – wobei theoretisch mit `Boost.Multithread` eine interessante und portable Alternative zur Verfügung steht. Allerdings würde eine Beschreibung dieser Bibliothek den Rahmen des Buches sprengen.
- Die Verwendung von Prozessen. Hierbei wird für jeden Client ein neuer (Server-)Prozess gestartet, jeder Client bekommt praktisch seinen eigenen Server.

Voraussetzung ist allerdings, dass Sie sich mit der Systemprogrammierung der entsprechenden Plattform auskennen. Schließlich müssen die einzelnen Prozesse kontrolliert werden.

Hinweis [«]

Wenn Sie mit den Techniken des Multithreadings und/oder der Prozessverwaltung auf den unterschiedlichen Systemen vertraut sind, könnten Sie auch hier wieder einen abstrakten Layer als portable Variante erstellen.

Neben diesen gibt es selbstverständlich eine Reihe weiterer Möglichkeiten, um mehrere Clients zu behandeln. Unter MS Windows könnten Sie hierfür die `WSA`-Routinen `WSAAsyncSelect()` oder `WSAEventSelect()` verwenden. Bei Linux/UNIX hingegen würden sich auch asynchrone E/A-Routinen nach »POSIX«-Erweiterungen eignen.

»select()« – eine portablere Alternative

Neben den eben beschriebenen Möglichkeiten, die Sie verwenden können, um mehrere Clients zu behandeln, soll hier auf die Möglichkeit mit der Funktion `select()` etwas genauer eingegangen werden. Diese Funktion ist sowohl auf MS- als auch auf Linux/UNIX-Systemen vorhanden und somit ein geeigneter Kandidat für eine portablere Lösung.

Das Problem bei einem Server, wie Sie ihn bisher verwendet haben, besteht darin, dass dieser immer nur auf einen Socket-Deskriptor wartet und auch immer über einen Socket-Deskriptor Daten empfangen bzw. gesendet werden. Wird beim Server zum Beispiel `revc()` aufgerufen, blockiert dieser Aufruf den Socket-Deskriptor so lange, bis der Client wirklich Daten an diesen gesendet hat. Man kann zwar das Blockieren auch dadurch umgehen, dass man den Socket-Deskriptor als nicht blockierend einrichtet (beispielsweise mit `fcntl()`), aber man sollte bedenken, dass hierbei ständig überprüft wird, ob an einem Socket Daten vorliegen. Das heißt, in einer Schleife wird ständig überprüft, was die CPU unnötig belastet. Mit der Funktion `select()` können Sie den Socket-Deskriptor so einrichten, dass nur dann CPU-Zeit benötigt wird, wenn auch wirklich Daten an einem Socket-Deskriptor vorliegen.

Hinweis [«]

Dieser Abschnitt sollte nicht den Eindruck erwecken, die Funktion `select()` sei eine Routine, die sich nur zur Netzwerkprogrammierung eignet. `select()` kann überall dort eingesetzt werden, wo auch Deskriptoren verwendet werden bzw. synchrones Multiplexing verwendet werden soll. Des Weiteren lassen sich mit `select()` auch hervorragend sogenannte *Timeouts* einrichten.

Hier folgt nun die Syntax zur Funktion `select()` unter Linux/UNIX:

```
// entsprechend POSIX 1003.1-2001
#include <sys/select.h>

// entsprechend früheren Standards
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select( int n,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout );
```

Und hier die ähnliche Syntax unter MS Windows:

```
int select( int n,
            fd_set FAR * readfds,
            fd_set FAR * writefds,
            fd_set FAR * exceptfds,
            const struct timeval FAR * timeout );
```

Mit dem ersten Parameter `n` geben Sie die Größe der folgenden Menge an. Hierfür wird im Allgemeinen der Wert des höchsten (Socket-)Deskriptors plus eins angegeben. Sie sollten sich allerdings nicht darauf verlassen, dass hier automatisch eine aufsteigende und lückenlose Reihenfolge für die (Socket-)Deskriptoren vergeben wird. Welche Nummer der nächste (Socket-)Deskriptor verwendet, entscheidet immer noch das System. Daher empfiehlt es sich, jeden gesetzten (Socket-)Deskriptor mit dem zu vergleichen, der rein theoretisch der höchste ist.

Die nächsten drei Parameter sind Zeiger auf die `fd_set`, die zum Lesen, Schreiben oder auf Ausnahmen getestet werden. Sofern Sie einen der Parameter nicht verwenden wollen, können Sie hierfür `NULL` angeben. Drei getrennte Sets sind nötig, da man ja nicht alle (Socket-)Deskriptoren auf Lesen oder Schreiben testen möchte.

Der am häufigsten verwendete Parameter (wie auch im anschließenden Beispiel) ist `readfds`. Mit diesem Parameter wird überprüft, ob auf den (Socket-)Deskriptoren Daten zum Lesen vorhanden sind. Das Gegenstück dazu ist der Parameter `writefds` – hiermit können Sie die Beschreibbarkeit von (Socket-)Deskriptoren überprüfen, das heißt prüfen, ob ein Deskriptor bereit ist, eine Ausgabe anzunehmen (dies wird zum Beispiel gerne bei Pipes verwendet). Der dritte `fd_set`-Parameter `exceptfds` wird weitaus seltener verwendet. Dieser kann eingesetzt

werden, um zu überprüfen, ob bei einem (Socket-)Deskriptor irgendwelche besonderen Zustände (Ausnahmen) vorliegen. Dies wird zum Beispiel bei Out-of-band-Daten (`MSG_OOB`) verwendet (siehe Manual-Page zu `send()` und/oder `recv()`).

Nach dem Aufruf von `select()` wird diese Menge in Teilmengen der Filedeskriptoren verteilt, die die Bedingungen erfüllen.

Mit dem letzten Parameter können Sie ein Timeout, eine Zeit im Format von Sekunden (`tv_sec`) und Mikrosekunden (`tv_usec`), einrichten. Diese Zeit wird dann gewartet, bis eine bestimmte Bedingung eintritt. Sind Sie daran nicht interessiert, können Sie auch hier `NULL` angeben. Es gibt aber auch einen Nachteil, wenn sich `select()`vor Ablauf der festgelegten Zeit verabschiedet. `select()` gibt keine Auskunft darüber, wie lange tatsächlich gewartet wurde. Dazu muss extra eine Funktion wie beispielsweise `gettimeofday()` aufgerufen werden.

Die Funktion gibt die Anzahl der Filedeskriptoren zurück, die ihre Bedingung erfüllt haben – das heißt die Anzahl der (Socket-)Deskriptoren, die bereit sind. Wenn die Zeit abgelaufen ist (Timeout), wird `0` und bei einem Fehler des Funktionsaufrufs `select() -1` zurückgegeben.

Ein Problem bei `select()` ist, dass diese Funktion mit Bit-Feldern arbeitet und somit abhängig vom Betriebssystem ist. Die Bit-Feldgröße bei BSD beträgt 256 und unter Linux 1.024. Somit können auf BSD nur die ersten 256 und unter Linux 1.024 Deskriptoren angesprochen werden. Unter MS Windows kann dieser Wert sogar nur bis zu 64 (Deskriptoren) betragen. Wie viele Deskriptoren Sie tatsächlich pro Prozess verwenden können, ist mit der symbolischen Konstante `FD_SETSIZE` definiert. Natürlich hat es wenig Sinn, alle (Socket-)Deskriptoren zu überwachen. Zum Glück müssen Sie sich kaum um diese Menge kümmern, da Ihnen der Datentyp `fd_set` die Arbeit zum Speichern der (Socket-)Deskriptoren abnimmt und einige Makros den Zugriff darauf erleichtern. Hier die Makros, um die Mengen zu bearbeiten:

```
FD_ZERO(fd_set *set);
FD_SET(int element, fd_set *set);
FD_CLR(int element, fd_set *set);
FD_ISSET(int element, fd_set *set);
```

Die Makros lassen sich schnell erklären. `FD_ZERO()` macht aus der Menge `set` eine leere Menge, `FD_SET()` fügt `element` der Menge `set` hinzu, und `FD_CLR()` entfernt `element` aus der Menge `set`. Mit `FD_ISSET()` können Sie überprüfen, ob `element` in der Menge `set` vorkommt bzw. gesetzt ist.

Das folgende Beispiel, (wieder) ein einfacher TCP-Echo-Server, soll die Funktion `select()` demonstrieren. Beim Server begnügen wir uns damit, dass dieser nur die Zeichenketten auf dem Bildschirm ausgibt und dem Client nicht antwortet. Allerdings besteht der gravierende Unterschied darin, dass der Server nun mehrere Clients »gleichzeitig« behandeln kann, und zwar bis zu `FD_SETSIZE`-Clients. Sobald auch hier ein Client die Zeichenfolge »quit« sendet, entfernt der Server den Client bzw. den (Socket-)Deskriptor aus der Menge.

Im Beispiel wurde wegen der Übersichtlichkeit darauf verzichtet, die `select()`-Abhandlung in unserer Klasse `Socket` zu implementieren. In der Praxis wäre dies allerdings sehr sinnvoll, da die Verwendung von `select()` zu den etwas komplizierteren Teilen der Programmierung gehört. Allerdings benötigen wir noch zwei Methoden, mit denen wir die Nummer des Socket-Deskriptors ermitteln bzw. setzen können. Dies ist erforderlich, weil `select()` hierbei die Nummer des Socket-Deskriptors benötigt, um diesen in der Menge ein- bzw. auszutragen. Da diese Nummer zu den privaten Eigenschaften der Klasse gehört, bauen wir in der Header-Datei *socket.h* einfach diese beide Methoden, `set_m_sock()` und `get_m_sock()`, ein, die unser Problem lösen. Wir nehmen die Definition gleich in *socket.h* vor, wodurch die beiden Methoden implizit als *inline* gekennzeichnet sind. Hier folgt nun die Header-Datei *socket.h* mit den beiden neuen Methoden:

```
/* socket.h */
#ifndef SOCKET_H_
#define SOCKET_H_
#include <string>
#include <winsock.h>
#include <io.h>
using namespace std;

// Max. Anzahl Verbindungen
const int MAXCONNECTIONS = 5;
// Max. Anzahl an Daten, die auf einmal empfangen werden
const int MAXRECV = 1024;

// Die Klasse Socket
class Socket {
private:
   // Socketnummer (Socket-Deskriptor)
   int m_sock;
   // Struktur sockaddr_in
   sockaddr_in m_addr;

public:
   // Konstruktor
```

```
   Socket();
   // virtueller Destruktor
   virtual ~Socket();

   // Socket erstellen - TCP
   bool create();
   // Socket erstellen - UDP
   bool UDP_create();
   bool bind( const int port );
   bool listen() const;
   bool accept( Socket& ) const;
   bool connect ( const string host, const int port );
   // Datenübertragung - TCP
   bool send ( const string ) const;
   int recv ( string& ) const;
   // Datenübertragung - UDP
   bool UDP_send( const string, const string,
                  const int port ) const;
   int UDP_recv( string& ) const;
   // Socket schließen
   bool close() const;
   // WSAcleanup()
   void cleanup() const;
   bool is_valid() const { return m_sock != -1; }
   // für select()
   int get_m_sock() const { return m_sock; }
   void set_m_sock( int nr ) { m_sock = nr; }
};
// Exception-Klasse
class SockExcept {
private:
   string except;

public:
   SockExcept( string s ) : except( s ) {};
   ~SockExcept() {};
   string get_SockExcept() { return except; }
};
#endif
```

Der Server – »multi_server.cpp«

```
// multi_server.cpp
#include "socket.h"
#include <iostream>
```

```
using namespace std;

int main() {
   int i, ready, sock_max, max=-1;
   int client_sock[FD_SETSIZE];
   fd_set gesamt_sock, lese_sock;
   try {
      Socket sock1, sock2, sock3;
      sock1.create();
      sock1.bind(15000);
      sock1.listen();
      sock_max = sock1.get_m_sock();
      for(i=0; i<FD_SETSIZE; i++)
         client_sock[i]=-1;
      FD_ZERO(&gesamt_sock);
      FD_SET(sock1.get_m_sock(), &gesamt_sock);
      for (;;) {
         // Immer aktualisieren
         lese_sock = gesamt_sock;
         /* Hier wird auf die Ankunft von Daten oder
          * neuer Verbindungen von Clients gewartet */
         ready = select( sock_max+1, &lese_sock,
                         NULL, NULL, NULL );
         // Eine neue Client-Verbindung ... ?
         if( FD_ISSET(sock1.get_m_sock(), &lese_sock)) {
            sock1.accept(sock2);
            /* Freien Platz für (Socket-)Deskriptor
             * in client_sock suchen und vergeben */
            for( i=0; i< FD_SETSIZE; i++)
               if(client_sock[i] < 0) {
                  client_sock[i] = sock2.get_m_sock();
                  break;
               }
            // Mehr als FD_SETSIZE Client sind nicht möglich
            if( i == FD_SETSIZE )
               throw SockExcept(
                  "Server überlastet - zuviele Clients");
            /* Den neuen (Socket-)Deskriptor zur
             * (Gesamt-)Menge hinzufügen */
            FD_SET(sock2.get_m_sock(), &gesamt_sock);
            /* select() benötigt die höchste
             * (Socket-)Deskriptor-Nummer */
            if( sock2.get_m_sock() > sock_max )
               sock_max = sock2.get_m_sock();
            /* höchster Index für client_sock
```

```
            * für die anschließende Schleife benötigt */
           if( i > max )
              max = i;
           // ... weitere (Lese-)Deskriptoren bereit?
           if( --ready <= 0 )
              continue; //Nein ...
        } // if(FD_ISSET ...
        /* Ab hier werden alle Verbindungen von Clients auf
         * die Ankunft von neuen Daten überprüft */
        for(i=0; i<=max; i++) {
           string s;
           sock3.set_m_sock(client_sock[i]);
           if((sock3.get_m_sock()) < 0)
              continue;
           // (Socket-)Deskriptor gesetzt ...
           if(FD_ISSET(sock3.get_m_sock(), &lese_sock)) {
              // ... dann die Daten lesen
              sock3.recv(s);
              cout << "Nachricht empfangen: " << s << endl;
              cout.flush();
              if( s == "quit" ) {
                 sock3.close();
                 //aus Menge löschen
                 FD_CLR(sock3.get_m_sock(), &gesamt_sock);
                 //auf -1 setzen
                 client_sock[i]=-1;
                 cout << "Ein Client hat sich beendet"
                      << endl;
              }
              // Noch lesbare Deskriptoren vorhanden ...?
              if( --ready <= 0 )
                 break; //Nein ...
           }
        }
     } // for(;;)
  }
  catch (SockExcept& e) {
     cout << "Exception wurde ausgeworfen: "
          << e.get_SockExcept() << endl;
  }
  return 0;
}
```

Der Client – multi_client.cpp

Der Quellcode des Clients bedarf keiner speziellen Anpassung und macht nichts anderes, als einzelne Strings an den Server zu senden, bis im Client »quit« eingegeben wurde.

```
// multi_client.cpp
#include "socket.h"
#include <string>
#include <iostream>
#include <cstdlib>
using namespace std;

int main (int argc, char *argv[]) {
   if (argc < 2) {
      cout << "Usage: " << argv[0]
           << " ServerAdresse" << endl;
      exit (1);
   }
   string argv_1 = argv[1];
   try {
      Socket client;
      client.create();
      client.connect( argv_1, 15000 );
      string mes;
      do {
         cout << "Nachricht zum Versenden: ";
         getline( cin, mes, '\n' );
         client.send( mes );
      } while( mes != "quit" );
      client.close();
   }
   catch ( SockExcept& e ) {
      cout << "Eine Exception wurde abgefangen: "
           << e.get_SockExcept() << endl;
   }
   return 0;
}
```

Das Beispiel ausführen

Wenn Sie die beiden Anwendungen übersetzt haben, starten Sie zunächst die Server-Anwendung (zur einfacheren Übersicht habe ich die Fenster nummeriert):

```
[---  Fenster 1  ---]
$> ./multi_server
```

Der Server ist nun bereit, Daten von mehreren Clients zu empfangen. Starten Sie also als Nächstes eine Client-Anwendung:

```
[---  Fenster 2  ---]
$> ./multi_client 127.0.0.1
Nachricht zum Versenden: Hallo Server von Fenster1
Nachricht zum Versenden:
```

Wenn Sie eine Nachricht an den Server gesendet haben, beendet sich jetzt nicht, wie in den Beispielen bisher, die Client-Anwendung. Schließlich soll ja auch die Behandlung mehrerer Clients demonstriert werden. Starten Sie nun eine neue Konsole und darin eine neue Client-Anwendung:

```
[---  Fenster 3  ---]
$> ./multi_client localhost
Nachricht zum Versenden: Hallo Server von Fenster2
Nachricht zum Versenden:
```

Allein dass der zweite Client sich hier gleich ausführen lässt, beweist, dass mehrere Clients gleichzeitig zugelassen werden. Würden Sie kein `select()` verwenden, würde die Ausführung der zweiten Client-Anwendung in eine Warteschlange kommen, bis der Server mit dem ersten Client komplett fertig ist und über `Socket::accept()` bereit für weitere Anfragen. Sie können jetzt (fast) beliebig viele Clients starten und den Server mit Nachrichten überhäufen. Ein Blick auf de Server zeigt jetzt Folgendes:

```
[---  Fenster 1  ---]
$> ./multi_server
Nachricht empfangen: Hallo Server von Fenster1
Nachricht empfangen: Hallo Server von Fenster2
```

Jetzt wollen wir einen Client beenden:

```
[---  Fenster 2  ---]
$> ./multi_client 127.0.0.1
Nachricht zum Versenden: Hallo Server von Fenster1
Nachricht zum Versenden: Ich beende mich jetzt gleich
Nachricht zum Versenden: quit
$>
```

Auf dem Server wird Folgendes ausgegeben:

```
[---  Fenster 1  ---]
$> ./multi_server
Nachricht empfangen: Hallo Server von Fenster1
Nachricht empfangen: Hallo Server von Fenster2
```

```
Nachricht empfangen: Ich beende mich jetzt gleich
Ein Client hat sich beendet
```

Jetzt wollen wir auch noch unseren letzten Client beenden:

```
[---  Fenster 3  ---]
$> ./multi_client localhost
Nachricht zum Versenden: Hallo Server von Fenster2
Nachricht zum Versenden: Ich mach mich jetzt auch weg
Nachricht zum Versenden: quit
$>
```

Ein letzter Blick zurück auf den Server:

```
[---  Fenster 1  ---]
$> ./multi_server
Nachricht empfangen: Hallo Server von Fenster1
Nachricht empfangen: Hallo Server von Fenster2
Nachricht empfangen: Ich beende mich jetzt gleich
Ein Client hat sich beendet
Nachricht empfangen: Ich mach mich jetzt auch weg
Ein Client hat sich beendet
```

Der Server läuft jetzt natürlich noch weiter und wartet auf Anfragen der Clients.

9.8 Weitere Anmerkungen zur Netzwerkprogrammierung

Zwar haben Sie jetzt viele grundlegende Dinge zur Netzwerkprogrammierung erfahren, doch ist dieses Gebiet ein sehr umfangreiches und leider auch gefährliches. Daher will ich noch einige Anmerkungen hinzufügen, was Sie noch beachten müssen, bevor Sie eigene Programme mit Netzwerkfunktionalität schreiben.

9.8.1 Das Datenformat

In den Beispielen, die Sie hier erstellt haben, wurden lediglich Zeichenketten verschickt. Meistens liegen die Daten aber in keinem so bequemen Format vor. Wenn Sie zum Beispiel Ganzzahlen oder Gleitkommazahlen versenden wollen, verwenden Sie am besten die String-Streams (siehe Abschnitt 7.2.3, »Klassen für String-Streams«).

Auch bei binären Strukturen empfiehlt es sich, die komplette Struktur in eine Zeichenkette zu konvertieren, bevor Sie diese versenden. Auf der anderen Seite müssen selbstverständlich ebenfalls bestimmte Vorkehrungen getroffen werden.

Letztendlich entscheiden Sie, wie die Daten zwischen Client und Server hin- und hergeschickt werden. Allerdings sollten Sie bedenken, dass Sie nicht immer wissen, auf was für einen Rechner die Daten übertragen werden. Schicken Sie von einem Little-Endian-Rechner einen Integer an ein Big-Endian-System, sind Probleme vorprogrammiert. Oder was ist, wenn Sie von einem 64-Bit-Rechner einen Integer an einen 32-Bit-Rechner schicken? Sie wissen also nie ganz genau, welche Größe die Datentypen `int`, `long` und `short` auf der Gegenseite haben. Theoretisch steht Ihnen hierfür in der Header-Datei <*limits*> die Template-Klasse `numeric_limits` zur Verfügung (siehe Abschnitt 7.3.4, »Grenzwerte von Zahlentypen«). Dies gilt allerdings nur dann, wenn Sie eine Server- und eine Client-Anwendung erstellen und entsprechend anpassen. Häufig erstellen Sie aber nur eine Client oder eine Server-Anwendung, die Daten von der Gegenstelle abholt oder eben empfängt.

Daher die Empfehlung: Senden Sie numerische Daten immer im Textformat an die Gegenseite. Natürlich setzt dies voraus, dass die Gegenseite denselben Zeichensatz verwendet. Wenn Sie einen String an einen Rechner schicken, auf dem sich nur japanische Schriftarten befinden, kommt nichts dabei heraus. Ebenso müssen Sie die im Deutschen vorkommenden Umlaute berücksichtigen, die auf vielen Rechnern nicht richtig dargestellt werden können. Der Nachteil ist, dass Bandbreite verschwendet wird. Eine 64-Bit-Nummer zum Beispiel kann nämlich über 20 Zeichen lang sein, während im Binärformat nur acht Zeichen dafür benötigt werden.

9.8.2 Der Puffer

Bisher mussten Sie sich kaum um die Pufferung der Daten kümmern. Sie mussten in der Regel nur angeben, wie groß der Puffer sein sollte. In der Netzwerkprogrammierung müssen Sie sich nun selbst darum kümmern. Mit den Funktionen `send()`/`sendto()` und `recv()`/`recvfrom()` können bei den Sockets weniger Bytes ein- bzw. ausgegeben als angenommen werden. Das Problem ist, dass das System (der Kernel) für das Socket eine bestimmte Puffergröße vorgibt. Das bedeutet: Wenn der Puffer voll ist, liest `recv()`/`recvfrom()` bzw. schreibt `send()`/`sendto()` aus diesem bzw. in diesen Puffer – selbst wenn noch nicht alle gewünschten Daten ausgelesen bzw. geschrieben wurden.

Wenn Sie die Daten in Form eines `char`-Arrays mit einfachem Text übertragen, dürften Sie keine Probleme mit der Pufferung bekommen, sofern Sie einen String mit `\0` abschließen. Sobald allerdings binäre Daten übertragen werden sollen, gibt es Probleme damit. Bei binären Daten können Sie sich nicht darauf verlassen, dass diese mit einem `\0` abgeschlossen werden, weshalb Sie sich hierbei selbst um das letzte Zeichen kümmern müssen.

Welche Puffergröße Sie verwenden, bleibt Ihnen überlassen und hängt vom Anwendungsfall ab. Allerdings macht ein byteweiser Puffer genauso wenig Sinn wie ein überdimensional großer Puffer. Es hat sich bewährt, eine Puffergröße von 512 oder 1024 KB zu verwenden.

[»] **Hinweis**

Der Puffer ist Ihr wichtigstes Kommunikationswerkzeug, mit dem Sie Daten austauschen können. Wenn Sie etwas hineinschreiben, sollten Sie daher immer bedenken, wie dies auf der anderen Seite wieder herauskommt, und eventuell auch überprüfen, was herauskommt. Was häufig nicht funktioniert, ist die Art und Weise, wie die Daten beim Empfänger ankommen. Nicht selten ist es ein nicht terminierter String, der für Zeichensalat sorgt.

9.8.3 Portabilität

Sie haben in diesem Kapitel gesehen, wie man mit einem abstrakten Layer eine portable Anwendung (nicht nur) für die Netzwerkprogrammierung erstellen kann. Mit Linux/UNIX und MS Windows haben Sie in den Beispielen eine recht große Zielgruppe eingeschlossen. Bedenken Sie allerdings, dass es auch noch andere Systeme gibt. Gemeint sind Systeme wie QNX oder SGI IRIX. Zwar sind die Unterschiede der allgemeinen Socketprogrammierung nicht allzu gravierend, dennoch müssen Sie sich auch schlaumachen, welche Differenzen es gibt.

9.8.4 Von IPv4 nach IPv6

Da IPv6 noch nicht eingeführt wurde (und eine Einführung auch noch nicht in Sicht ist), wurden die Eigenheiten in den vorangegangenen Abschnitten nicht näher behandelt. Allerdings gibt es hierzu auch nicht viel zu berichten. Daher folgt nun eine kurze Zusammenfassung, wie Sie Ihre Anwendungen von IPv4 nach IPv6 portieren könnten.

Konstanten

Die IPv4-Konstanten `AF_INET` bzw. `PF_INET` wurden durch `AF_INET6` bzw. `PF_INET6` ersetzt. Es muss hierbei eigentlich nur die Konstante um eine `6` erweitert werden. Es ist auch kein Fehler, wenn Sie bei einer IPv4-Software gleich die neuen Konstanten verwenden, da ein Programm, das auf IPv6 portiert wurde, auch weiterhin auf IPv4-Rechnern läuft (vorausgesetzt, der Rechner ist *dual-stacked*, was in Zukunft bei IPv6-fähigen Rechnern immer der Fall sein sollte).

Verändert hat sich auch die Konstante `INADDR_ANY`, die beim Binden von Sockets an einen Port angegeben wird und die besagt, dass Pakete von jedem Interface angenommen werden. Ein wenig ungewöhnlich ist, dass die neue Konstante

kleingeschrieben wird – `in6addr_any`. Der Grund hierfür: Die alte Struktur `in_addr` bestand nur aus einem `unsigned long int s_addr`, und somit war die Konstante `INADDR_ANY` auch nur eine Zahl. Da die Adresse bei IPv6 128 Bit breit ist, ist dies nicht mehr möglich (da kein portabler Datentyp mit dieser Breite existiert), weshalb es sich nun um ein Array handelt:

```
struct in6_addr {
   union {
      uint8_t  u6_addr8[16];
      uint16_t u6_addr16[8];
      uint32_t u6_addr32[4];
   } in6_u;

#define s6_addr    in6_u.u6_addr8
#define s6_addr16  in6_u.u6_addr16
#define s6_addr32  in6_u.u6_addr32
};
```

Strukturen

Nachdem `in_addr` durch `in6_addr` ersetzt wurde (siehe letzten Abschnitt), ist es auch nötig, die Struktur `sockaddr_in` anzupassen:

```
struct sockaddr_in6 {
   sa_family_t sin6_family     // Address family - AF_INET6
   in_port_t sin6_port;        // Transport layer port #
   uint32_t sin6_flowinfo;     // IPv6 flow information
   struct in6_addr sin6_addr;  // IPv6 address
   uint32_t sin6_scope_id;     // IPv6 scope-id
};
```

Abgesehen davon, dass die Adressstruktur verändert wurde, wurden auch die zusätzlichen Strukturvariablen `sin6_flowinfo` und `sin6_scope_id` hinzugefügt.

Funktionen

Der Großteil der Socket-API-Funktionen ist gleich geblieben. Verändert (hinzugefügt) wurden lediglich die meisten Adressauflösungs- und Konvertierungsfunktionen. So werden die Funktionen `inet_aton()` bzw. `inet_ntoa()` durch die Funktionen `inet_pton()` bzw. `inet_ntop()` ersetzt. Da diese neuen Funktionen jetzt nicht mehr mit Zahlen operieren, sondern mit den konkreten Adressstrukturen (z. B. `in6_addr`), unterstützen sie auch beliebige Adressfamilien.

Noch wichtiger sind die neu hinzugekommenen Funktionen `getaddrinfo()` und `getnameinfo()`. Diese wurden als Ersatz für die Funktionen `gethostbyname()`/`gethostbyaddr()` und `getipnodebyname()`/`getipnodebyaddr()` eingeführt und

haben den Vorteil, dass sie direkt `sockaddr`-Strukturen bearbeiten. Des Weiteren wurde noch die Funktion `gethostbyname2()` hinzugefügt, wobei es sich allerdings nur um eine reine GNU-Extension handelt!

[»] **Hinweis**

All diese Funktionen stehen Ihnen übrigens auch schon für IPv4 zur Verfügung, weshalb es nicht falsch sein kann, diese jetzt schon zu verwenden, um eine mögliche spätere Portierung zu erleichtern.

9.8.5 RFC-Dokumente (Request for Comments)

Sie wollen einen HTTP-, einen FTP- oder einen SMPT-Server bzw. einen Client erstellen, der damit kommuniziert, und wissen nicht, wo Sie anfangen sollen. Dies ist eine beliebte Frage in den Foren. Wie Sie bereits erfahren haben, findet die Kommunikation zwischen dem Server und dem Client mit Protokollen statt (vergleichbar mit den verschiedenen Sprachen dieser Welt). Ein Webclient und ein Webserver zum Beispiel unterhalten sich anders als ein Mailclient und ein Mailserver. All diese Standardprotokolle werden unter dem Namen *RFC (Request for Comments)* gesammelt. RFCs sind eine Reihe von technischen Dokumentationen zum Internet, die ihren Ursprung zu ARPANET-Zeiten 1969 hatten. Einen sehr großen Fundus zur RFC-Sammlung finden Sie im Internet unter *http://www.ietf.org/* (The Internet Engineering Task Force).

9.8.6 Sicherheit

Und das Wichtigste zum Schluss: Mit den Kenntnissen in der Netzwerkprogrammierung in C haben Sie allerdings auch das gefährlichste Kapitel in C kennengelernt. Die meisten Programme, die angegriffen werden, sind nicht Ihr Editor oder Ihre Entwicklungsumgebung, sondern die Programme, die mit dem Netz verbunden und somit meistens auch für alle erreichbar sind. Ein Buffer Overflow bei einer Netzwerk-Anwendung kann sehr böse Folgen haben (beliebtes Beispiel ist der Internet Explorer alias Internet Exploiter).

Des Weiteren sollten Sie beachten, dass die Daten, die Sie übers Netzwerk versenden, jederzeit abgefangen werden können. Daher empfiehlt es sich bei sicherheitsrelevanten Daten (beispielsweise Kundendaten), diese verschlüsselt zu versenden. Die Daten können zwar weiterhin abgefangen werden, aber bei einer guten Verschlüsselung sind sie für den »Sniffer« nicht mehr lesbar – es sei denn, er kann die Verschlüsselung knacken.

9.8.7 Fertige Bibliotheken

Häufig wird die Frage gestellt, ob es fertige C++-Bibliotheken für die Netzwerkprogrammierung gibt. In diesem Zusammenhang werden in der Praxis oft die beiden Bibliotheken `socket++` und `cppsocket` erwähnt. Beide Bibliotheken haben den Vorteil, dass sie auf den gängigsten Systemen vorhanden und vor allem gut und häufig erprobt sind. Da sich die Bezugsquellen dieser Bibliotheken häufig ändern können, sollte Ihnen Google hier weiterhelfen.

Wenn Sie dieses Buch von Anfang an durchgearbeitet haben, stehen Ihnen nun Tür und Tor für eigene Projekte offen. Meistens kommt jetzt schnell die Frage nach der grafischen Programmierung auf. Gerade für den Einsteiger kann es dabei recht unübersichtlich werden.

10 GUI- und Multimediaprogrammierung in C++

Als Erstes sollte man sich Gedanken darüber machen, welche Art von grafischer Programmierung (kurz GUI – *Graphical User Interface*) man verwenden will. Es gibt zwar diesbezüglich keinen Standard, aber ich persönlich unterscheide zwischen zwei Formen:

- GUI-Programmierung – damit können Sie echte fensterbasierende Anwendungen wie einen Text-Editor, einen Webbrowser oder einen E-Mail-Client programmieren.
- Grafik- und Multimediaprogrammierung – darunter fällt das Programmieren von Spielen, Demos oder Grafikanwendungen. Auch andere multimediale Dinge wie Musik und Videos gehören in diesen Bereich.

Auf den folgenden Seiten werde ich Ihnen zunächst einen Überblick geben und anschließend auf die Bibliothek von `wxWidgets` etwas genauer eingehen.

Hinweis [«]

Hierbei muss ich allerdings anmerken, dass es sich beim anschließenden Praxisteil (zu `wxWidgets`) wieder nur um eine Einführung handelt. Bei vielen dieser Bibliotheken ist das Manual bereits mehrere hundert Seiten dick. Außerdem finden Sie auf der Buch-CD die ein oder andere Dokumentation.

10.1 GUI-Programmierung – Überblick

Die Programmierung von grafischen Oberflächen (GUI) kann man in eine tiefere (*Low-Level*) und eine höhere (*High-Level*) Ebene einteilen.

10.1.1 Low-Level

Mit Low-Level bezeichne ich die tiefste Ebene der »grafischen« Programmierung. Hier müssen Sie sich um alle Details selbst kümmern. Sie wollen praktisch jeden Punkt, jede Linie und jedes kleine Detail (genauer die *Grafik-Primitiven*) selbst zeichnen. Gewöhnlich greifen Sie dabei auf die systemnächste Grafikschnittstelle des Betriebssystems zu. Unter MS Windows wäre dies die Win32-API und unter Linux/UNIX die Xlib (auch als X11 bekannt). Fast alle höheren Grafikbibliotheken bauen auf dieser Ebene auf (auch die objektorientierten). Häufig wird hierbei als Programmiersprache C verwendet.

Sehr gut geeignet ist diese systemnahe Grafikschnittstelle zum Entwickeln einer eigenen Spielebibliothek oder eines Window-Managers (Fenster-Managers), mit der bzw. dem Sie Ihrer Oberfläche ein eigenes Look and Feel verpassen können (beispielsweise KDE oder GNOME unter Linux).

Die Nachteile bei der Verwendung dieser niedrigen Ebene sind eine sehr lange Einarbeitungszeit und ein langer und komplizierter Quellcode. Außerdem sind gute Mathematikkenntnisse erforderlich, und das Ganze ist systemabhängig – also für Anfänger nur bedingt geeignet.

Natürlich haben Sie hierbei den Vorteil, dass Sie eine eigene, schlanke Bibliothek entwickeln können, die Ihren Vorstellungen entspricht. Des Weiteren lernen Sie dabei, wie grafische Oberflächen aufgebaut sind, was Ihnen auf jeden Fall beim Verwenden einer anderen grafischen Bibliothek hilft. Dabei werden Sie feststellen, dass das Prinzip einer grafischen Bibliothek immer dasselbe bleibt.

10.1.2 High-Level

Grafikbibliotheken der höheren Ebene basieren meistens auf der niedrigeren Ebene (Win32 bzw. X11). Bei solchen Bibliotheken können Sie vieles mit einem einzigen Funktionsaufruf erledigen. Alle unangenehmen Arbeiten werden Ihnen abgenommen. Häufig wird dabei auch Systemabhängiges versteckt bzw. eine Cross-Plattform-Bibliothek dafür angeboten. Damit lassen sich viele Bibliotheken der höheren Ebene auf mehreren Systemen verwenden.

Vorwiegend werden solche Bibliotheken verwendet, um die auf dem System basierenden, typischen Look-and-Feel-Anwendungen wie Outlook bzw. Kmail zu entwickeln. Hierbei ist die Programmiersprache nicht mehr von primärer Bedeutung, da es häufig Schnittstellen für mehrere Sprachen gibt (für C bzw. C++ gibt es generell immer eine Schnittstelle). In der Regel werden einer solchen Bibliothek auch eine Dokumentation und Beispiele mitgegeben, die die Einarbeitung wesentlich erleichtern.

Der Nachteil solch höherer Bibliotheken ist, dass die Anwendungen manchmal etwas »fett« werden und viel unnötigen Ballast mitführen. Dass es aber auch anders geht, beweist die Bibliothek FLTK, mit der sich ein recht schlankes GUI-Programm erstellen lässt.

Zudem überwiegt der Vorteil, dass die Erstellung einer Anwendung wesentlich schneller und komfortabler von der Hand geht (ein wenig Einarbeitungszeit vorausgesetzt). Dass sich der Programmierer mit seinem Werkzeug (besonders dem Compiler und Linker) auskennt, ist allerdings immer Voraussetzung.

RAD-Tools

Noch einfacher wird es dem Programmierer mit sogenannten *RAD-Tools (RAD = Rapid Application Development)* oder auch dem GUI-Designer gemacht. Sie ermöglichen es, fast ohne Programmierkenntnisse eine grafische Anwendung zu erstellen.

Aber auch wenn diese RAD-Tools immer besser werden und dem Programmierer viel Arbeit abnehmen, sind weiterhin Programmierkenntnisse notwendig, wenn es um Details geht. Ein Mix aus beidem (RAD-Tools und Programmierung »von Hand«) führt wohl zum besten Ergebnis.

10.1.3 Überblick über plattformunabhängige Bibliotheken

Hinweis [«]

Hierbei handelt es sich nicht um einen Überblick über alle auf der Welt vorhandenen Bibliotheken zur GUI-Programmierung, sondern um Bibliotheken, die für umfangreichere C++-Projekte (High-Level wohlgemerkt) mit Gewinn eingesetzt werden können. Außerdem sind alle hier vorgestellten Bibliotheken kostenlos erhältlich (mehr zu den Lizenzen der einzelnen Bibliothek erfahren Sie auf den offziellen Webseiten). Sollte jemand der Meinung sein, dass ich hierbei eine tolle Bibliothek übersehen habe, würde ich mich über einen Hinweis sehr freuen.

- FLTK – FLTK (kurz für *Fast and Light Toolkit;* ausgesprochen »fulltick«) ist eine Cross-Plattform C++ GUI Toolkit für Linux/UNIX, Microsoft Windows und Mac OS. FLTK unterstützt die komplette Funktionalität, die man sich von einem modernen GUI erwartet, und unterstützt sogar 3D-Grafik über OpenGL (und dessen built-in GLUT – ursprünglich wurde FLTK ja für die 3D-Grafikprogrammierung entwickelt). FLTK ist freie Software und enthält FLUID, einen eigenen GUI-Designer (FLUID steht für *FLTK User Interface Designer)*. Anders als Bibliotheken wie Qt oder wxWidgets stellt FLTK ausschließlich User-Interface-Funktionalität zur Verfügung und ist daher sehr kompakt. Deshalb wird FLTK üblicherweise statisch in die Applikationen gebunden, was die erwei-

terte LGPL-Lizenz von FLTK gestattet. Trotz statischer Bindung ist ein »Hello World«-Programm mit FLTK nur etwas über 100 KB groß – was für eine GUI-Anwendung beachtlich ist (gewöhnlich verbraten solche Programme Megabytes!). Offizielle Webseite: *http://www.fltk.org/*

- Fox – Fox ist ein weiteres auf C++ basierendes Toolkit für die Entwicklung von GUI-Anwendungen. Die Stärken dieses Toolkits sind dabei eine einfache Verwendbarkeit und eine sehr effiziente Ausführung der Anwendung (auch das ist nicht selbstverständlich). Fox ist ein sehr ausgereiftes und mittlerweile lang bewährtes Toolkit, das neben den üblichen GUI-Features über eine sehr umfangreiche Sammlung an Extras verfügt (beispielsweise Drag & Drop, Selectionen, OpenGL-Widgets, Tooltipps etc.). Auch die Plattformen, die das Fox-Toolkit mittlerweile unterstützt, sind beachtlich (Linux, FreeBSD, SGI IRIX, HP-UX, IBM AIX, SUN Solaris, DEC/Compaq Tru64 UNIX, alle MS Windows-Systeme wie Windows 9x, Windows NT, Windows ME und Windows 2000). Offizielle Webseite: *http://www.fox-toolkit.org/*
- gtkmm (gtk-) – gtkmm ist die offizielle C++-Schnittstelle für die beliebte und sehr populäre GUI-Bibliothek GTK+ mit allen typischen Vorteilen von C++ (typsichere Callbacks, das Erweitern von Widgets durch Vererbung etc.) Neben der üblichen Erstellung der Anwendungen per Code steht Ihnen auch hierbei mit GLADE ein eigener GUI-Designer zur Verfügung (basierend auf der Bibliothek libglademm). Die Liste der Cross-Plattformen für gtkmm ist ebenfalls beeindruckend: Linux (gcc), FreeBSD (gcc), NetBSD (gcc), Solaris (gcc, Forte), Win32 (gcc, MSVC++ .Net 2003), Mac OS X (gcc). Die offizielle Webseite: *http://gtkmm.sourceforge.net/*

[»] **Hinweis**

Wenn Sie mehr zur Programmierung mit Qt wissen wollen, kann ich Ihnen mein Buch »Qt 4 – GUI-Entwicklung mit C++« aus demselben Verlag empfehlen.

- Qt – Qt ist eines der führenden Frameworks für die Entwicklung von nativen Cross-Plattform-Applikationen. Die Qt-API und deren Tools sind auf allen Plattformen (Microsoft-System, Mac OS X, Linux, Solaris, HP-UX, IRIX, AIX und weitere UNIX-Varianten) vorhanden. Dieses Framework geht sogar so weit, dass sich der Entwickler eigentlich gar nicht mehr um die plattformspezifischen Dinge kümmern muss, sondern sich ganz auf die API konzentrieren kann – eine echte plattformunabhängige API. Auch die Features, die Qt anbietet, übertreffen alle anderen Toolkits bei weitem (zum Beispiel volle 2D-Grafik-API mit Rotationen, Skalieren etc., Metafile Support, gleicher Code für die Ausgabe vom Bildschirm und Drucker, Look and Feel variabel anpassbar etc.). Bei all diesen Features ergibt sich allerdings der Nachteil, dass der Überblick

über diese Funktionsvielfalt leicht verloren geht. Man sollte außerdem beachten, dass `Qt` nur für den nicht kommerziellen Einsatz kostenlos ist. Offizielle Webseite: *http://www.trolltech.com/*

- `wxWidgets` – `wxWidgets` ist eine einfach zu verwendende, aber sehr mächtige API, um eigene GUI-Applikationen auf den unterschiedlichsten Plattformen (Linux/UNIX, Mac OS, Windows etc.) zu erstellen. `wxWidgets` lässt sich sehr gut mit dem mächtigen MFC (von Microsoft) vergleichen – nur eben plattformunabhängig. Neben den üblichen Features eines Toolkits bietet `wxWidgets` folgende Besonderheiten an: Online-Hilfe, Netzwerkprogrammierung, Streams, Zwischenablage, Drag & Drop, Multithreading, Bilder laden und speichern in unterschiedlichen Formaten, Datenbank-Unterstützung (MySQL, SQLite etc.), HTML-Betrachter und Drucker, virtuelles Filesystem, OpenGL-Unterstützung und noch vieles mehr. Dieses Toolkit ist einzigartig und zudem kostenlos. Außerdem ist es vielfach erprobt und sehr gut dokumentiert. Ich war von diesem Toolkit so begeistert, dass ich es im Praxisteil aufgenommen habe. Offizielle Webseite: *http://www.wxwidgets.org/*

10.1.4 Überblick über plattformabhängige Bibliotheken

Es ist in der Tat sehr erfreulich, dass sich immer mehr Entwickler von Bibliotheken entscheiden, eine GUI-Cross-Plattform-Bibliothek zu schreiben. Aber es gibt auch einige Bibliotheken, darunter der Platzhirsch `MFC`, die nicht (oder zumindest nicht ohne Mitwirkung anderer Tools) auf mehreren Plattformen vorhanden sind, aber einfach nicht ignoriert werden können.

- `MFC` *(Microsoft Foundation Class)* – `MFC` ist immer noch ein häufig eingesetztes Toolkit in der GUI-Entwicklung unter Microsoft-Systemen. Zwar ist es über Add-one oder 3rd-Party-Tools möglich, auch unter Linux/UNIX bzw. Mac OS auf `MFC` zurückzugreifen – doch ist dies in der Praxis eher selten der Fall. Schließlich gibt es viele freie und kostenlose Alternativen. `MFC` ist eine kommerzielle Bibliothek. Offizielle Webseite: *http://www.microsoft.com*
- `OWL` (*Object Windows Library*) – bevor `MFC` gekommen ist, war `OWL` von Borland das am meisten verwendete Toolkit unter MS Windows. Heute wird `OWL` allerdings kaum noch verwendet. Erwähnt wird es hier, weil es immer noch viele Programme gibt, die damit erstellt wurden. Offizielle Webseite: *http://www.borland.com*
- `VCL` (*Visual Component Library*) – `VCL` ist die aktuelle Bibliothek von Borland/Inrise für MS Windows, die heute bei den aktuellen Borland-Compilern zum Einsatz kommt. `VCL` ist daher auch eine kommerzielle Bibliothek. Offizielle Webseite: *http://www.borland.com*

10.2 Multimedia- und Grafikprogrammierung – Überblick

C++ gilt immer noch als die Programmiersprache für Spiele-, Grafik- und Multimediaanwendungen. Allerdings sind diese Themen zum Teil sehr komplex und erfordern neben den Kenntnissen zu den einzelnen Bibliotheken häufig auch genauere mathematische Kenntnisse (besonders bei der Spieleprogrammierung). Trotzdem seien hier einige Bibliotheken erwähnt, die mittlerweile nicht mehr wegzudenken sind.

10.2.1 Überblick über plattformunabhängige Bibliotheken

Mittlerweile gibt es viele Bibliotheken, die unabhängig von der Plattform sind. Programmierer aus aller Welt bemühen sich heutzutage, Bibliotheken für die gängigsten Systeme zu entwickeln. Daher seien hier einige der geläufigsten erwähnt.

- `OpenGL` – die wohl bekannteste Alternative zur DirectX dürfte `OpenGL` sein (die leider immer mehr an Boden verliert gegenüber DirectX). `OpenGL` (*Open Graphic Library*) ist eine Schnittstelle (*API = Application Programming Interface*), die gewöhnlich zum Rendern einer Szene verwendet wird bzw. zum Erzeugen von einfachen und komplexen 3D-Grafiken (natürlich auch 2D). Mit ca. 250 Routinen ist die Bibliothek recht einfach gehalten. Die meisten dieser Routinen werden direkt in der Grafikkarte ausgeführt (was einen beachtlichen Geschwindigkeitsgewinn mit sich bringt). Komplizierte Berechnungen werden von der API übernommen, so dass auch der Mathematik-Laie beachtliche Ergebnisse erzielen kann.

  Zwar bietet die `OpenGL`-API schon eine ganze Menge beachtlicher Funktionen, die meisten sind allerdings recht umständlich. Häufig muss erheblicher Aufwand betrieben werden, um zum Beispiel ein einfaches Dreieck auf dem Fenster zum Rotieren zu bringen. Hinzu kommt noch ein systemabhängiger Aufwand für Ereignisse und Fenster. Aus diesem Grund wurden viele nützliche Bibliotheken geschrieben, die auf der (Low-Level-)API von `OpenGL` aufbauen und dem Programmierer viel Arbeit abnehmen (insbesondere den systemabhängigen Kram). Einige der bekanntesten Bibliotheken sind:

  - `GLU` *(OpenGL Utility Library)* – basierend auf der Low-Level-Schnittstelle von `OpenGL` enthält diese Bibliothek viele verschiedene Funktionen zu den Matrizen für spezielle Ansichten und Projektionen, Polygons und Rendering Surfaces – alles Routinen, die sich mit der OpenGL-Bibliothek recht mühevoll erstellen lassen, besonders wenn man mit diesem Gebiet (noch) nicht vertraut ist. Somit kann man mit `GLU` Funktionen benutzen, deren

Verwendung man kennt, von denen man aber nicht weiß, wie man sie in der Praxis realisieren kann. GLU-Funktionen beginnen immer mit dem Präfix glu.

- Für jedes Window-System gibt es außerdem eine Bibliothek, die neben den Systemfunktionen selbst auch das OpenGL-Rendering unterstützt. Bei Systemen, die das X-Window-System verwenden, ist dies beispielsweise die GLX-Bibliothek. Alle GLX-Funktionen haben somit das Präfix glX. Unter MS Windows lautet diese Bibliothek WGL, so dass hier die Funktionen mit dem Präfix wgl beginnen. Selbst für IBM OS/2 existiert mit PGL (Präfix pgl) eine solche systemeigene Bibliothek.
- GLUT *(OpenGL Utility Toolkit)* – GLUT ist ein systemunabhängiges Toolkit von Mark Kilgard, das die komplexen Dinge der verschiedenen Window-Systeme (Win32-API, Xlib etc.) versteckt. Damit ist es möglich, den Quellcode auf den verschiedensten Window-Systemen ohne Probleme zu übersetzen. Und außerdem werden dabei keine tiefgreifenden Systemkenntnisse benötigt. Alle Funktionen dieser Bibliothek beginnen mit dem Präfix glut. Der Entwickler Mark Kilgard hat ein Buch herausgebracht, in dem u. a. diese Bibliothek näher beschrieben wird – »OpenGL Programming for the X Window System«.
- MesaGL – MesaGL ist eine freie Implementierung von OpenGL und eine reine Software-Emulation. Die aktuelle stabilste Version von MesaGL ist 6.0 und von der Webseite *www.mesa3d.org* zu beziehen. MesaGL ist im Gegensatz zu OpenGL frei von der ARB-Lizenzierung. Neben MesaGL gibt es selbstverständlich auch eine reine Microsoft-Implementierung von OpenGL auf Softwarebasis.

- Allegro – eine in C geschriebene kostenlose Bibliothek zur Erstellung von Computerspielen in C und C++ (und auch anderen Sprachen). Die Stärken von Allegro sind Funktionen und Datenstrukturen für 2D-Grafik, Tonausgabe, Benutzereingabe usw. Natürlich unterstützt diese Bibliothek auch einfachere 3D-Grafik. Sie hatte ihre Zeit in den frühen 90er Jahren, wo sie als zusätzliche Bibliothek für den DJGPP-Compiler (DOS-Version von GCC) erhältlich war. Zwar war es eine Zeitlang recht ruhig um Allegro, aber seit 2004 zeichnet sich wieder eine dynamische Weiterentwicklung ab. Natürlich bietet diese Bibliothek mit AllegroGL auch eine OpenGL-Anbindung an. Allegro ist nach wie vor eine sehr beliebte Bibliothek zum Einsteigen in die Spieleprogrammierung, da diese sehr einfach zu verwenden, auf allen Plattformen vorhanden und sehr gut dokumentiert ist. Ein Blick auf die Webseite von *http://www.allegro.cc* zeigt, dass mittlerweile unzählige Projekte mit Allegro verwirklicht wurden (vorwiegend kostenlos und offen im Quellcode).

- SDL – der *Simple DirectMedia Layer (SDL)* ist eine freie (LGPL-)Multimediabibliothek für verschiedene Plattformen. Die Bibliothek stellt eine abstrakte API für Grafik-, Sound- und Eingabegeräte bereit. Somit steht dem Entwickler eine mächtige API zur Spiele- und Multimediaentwicklung zur Verfügung. Durch die Unterstützung vieler verschiedener Plattformen ist bei der Entwicklung eine hohe Portabilität gewährleistet. SDL ist in C geschrieben und zeichnet sich besonders durch einen im Vergleich zu anderen APIs aus diesem Bereich (z. B. DirectX) recht kompakten Code aus, wodurch diese Bibliothek auch für Anfänger leicht zu erlernen ist. Es existieren viele Beispielprogramme, die zeigen, wie die Bibliothek verwendet wird. Für über zehn Sprachen existieren Anbindungen. Was den Funktionsumfang betrifft, muss man anmerken, dass der Simple DirectMedia Layer nicht direkt Funktionen für 3D-Grafik zur Verfügung stellt – was aber durch die Einbindungsmöglichkeiten von OpenGL ausgeglichen wird, so dass es mit der SDL auch möglich ist, 3D-Spiele wie Tux Racer zu programmieren. Ich habe dem Thema SDL in dem Buch »Linux-UNIX-Programmierung« auch ein Kapitel gewidmet – hierbei allerdings in C.

10.2.2 Überblick über plattformabhängige Bibliotheken

Im Abschnitt zuvor wurden einige sehr interessante plattformunabhängige Bibliotheken zur Entwicklung von Multimedia- und Grafikanwendungen aufgelistet. Trotzdem ist hier der Platzhirsch Microsofts DirectX, womit noch immer die meisten (auch kommerziellen) Anwendungen dieser Art erstellt werden.

DirectX ist eine ganze Sammlung von APIs für Multimediaprogramme (besonders Spiele). Die Verwendung der APIs beschränkt sich allerdings nur auf Windows-Plattformen und die Spielekonsole Xbox360. DirectX wird vorrangig zur Darstellung komplexer 2D- und 3D-Grafik benutzt. Es bietet aber auch Unterstützung für Audio, diverse Eingabegeräte (z.B. Maus, Joystick) und Netzwerkkommunikation.

Die aktuelle Version von DirectX (zurzeit Version 10.1) teilt sich in folgende APIs auf:

- DirectX Graphics – Unterstützung für 2D- und 3D-Grafik
- DirectSound – zur Wiedergabe und Aufnahme von Soundeffekten (unterstützt auch 3D-Sound = Raumklang). Künftig wird wohl DirectSound von XAudio2 (basierend auf der Xbox-360-Sound-API) abgelöst.
- DirectMusic – zur Wiedergabe von Musik (beispielsweise MIDI)
- DirectInput – Unterstützung für Eingabegeräte wie Maus, Tastatur, Joysticks (auch mit Force-Feedback-Effekten) etc.

- DirectPlay – für die Kommunikation von Multiplayerspielen, die auf mehreren Computern laufen (Netzwerkspiele, Online-Spiele).
- DirectShow – für die Verarbeitung von Video- und Audio-Dateien
- DirectSetup – damit kann der Programmierer bei der Installationsroutine überprüfen lassen, ob die richtige DirectX-Version vorhanden ist, und diese gegebenenfalls nachinstallieren lassen.
- DirectX Media Objects – entspricht DirectSound und DirectShow

Die aktuelle Version von DirectX ist zwar derzeit die Version 10.1, aber bis Ende 2009 – mit der Einführung von Windows 7 – wird die Version 11 kommen. Anders allerdings als beim Versionswechsel von 9 auf 10 wird dieses Mal auch die Version 11 auf Windows Vista laufen. Das SDK von DirectX11 kann allerdings schon seit November 2008 heruntergeladen werden.

10.3 GUI-Programmierung mit »wxWidgets«

Hinweis [«]

Aus Platzgründen finden Sie auf der Buch-CD einige Hinweise und Anleitungen, wie Sie Programme mit `wxWidgets` auf den unterschiedlichen Compilern und Systemen installieren, übersetzen und ausführen können.

10.3.1 Warum »wxWidgets«?

Sicherlich stellen Sie sich die Frage, warum ich hier `wxWidgets` für die Erstellung einer GUI-Applikation empfehle. Ganz einfach, weil `wxWidgets` alles bietet, was ein komplettes Toolkit ausmacht. Einige Vorteile von `wxWidgets` sind diese:

- Es ist ziemlich vollständig – neben der üblichen GUI gibt es noch sehr viele Utility-Klassen.
- Es wird immer noch weiterentwickelt und verbessert.
- Viele Compiler und Plattformen werden unterstützt (MS Windows, Linux, Mac OS, UNIX etc.).
- Plattformspezifischer Code wird versteckt. Der Quellcode, den Sie hierbei unter Linux erstellen, läuft somit auch ohne Änderungen unter MS Windows oder Mac OS.
- Es ist sehr gut dokumentiert.
- Es ist sowohl für die private als auch für die kommerzielle Anwendung kostenlos.

- Es benutzt immer die SDK der Plattform. Wenn Sie also ein Programm unter Linux übersetzen, dann haben Sie das typische Look and Feel von Linux. Übersetzen Sie eine Anwendung unter Windows, dann wird auch das Look and Feel von Windows eingesetzt.

Hinweis zu »wxWidgets«/»wxWindows«

Wahrscheinlich sind Sie bei Ihren Recherchen zu `wxWidgets` auf zwei Namen gestoßen. In der Tat hat `wxWidgets` vor dem Jahr 2004 noch `wxWindows` geheißen. Die Namensänderung ist Folge einer Aufforderung von Microsoft, da Microsoft in Großbritannien die Marke »Windows« besitzt. Rein rechtlich gab es allerdings keine Möglichkeit, dies durchzusetzen. Julian Smart, der Gründer von `wxWindows` (jetzt `wxWidgets`), hat sich dennoch entschieden, sein Projekt in `wxWidgets` umzubenennen (gegen eine kleine finanzielle Entschädigung, heißt es).

10.3.2 Das erste Programm – Hallo Welt

Ich möchte hier keine Zeit mit den Details – wie zum Beispiel der Geschichte – verschwenden. Dazu können Sie bei Bedarf die Dokumentation lesen.

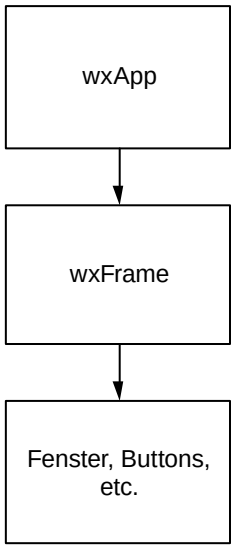

Abbildung 10.1 Grundgerüst einer »wxWidgets«-Applikation

Eine nackte `wxWidgets`-GUI-Applikation besteht mindestens aus den folgenden drei Teilen:

- einem Applikations-Objekt – dies ist eine Instanz der `wxApp`-Klasse.

- einem Frame-Objekt – dies ist eine Instanz der `wxFrame`-Klasse. Ein solches Frame kann wiederum Dinge enthalten wie eine Menübar, eine Statusbar etc.
- Dieses Frame kann dann weitere Objekte wie Buttons, Eingabeaufforderungen usw. enthalten.

Hinweis **[«]**

Als Autor stehe ich hier vor dem Problem, die einzelnen Bezeichnungen eines GUI ins Deutsche zu übersetzen oder englisch zu belassen. Ich habe mich dafür entschieden, möglichst die Begriffe zu verwenden, die am gängigsten sind.

Unser erstes Beispiel wird zunächst noch ein leeres Frame besitzen – eben ein leeres Fenster. Zunächst soll eine Header-Datei *base.h* erstellt werden, in der die Deklarationen der Klasse vorgenommen werden:

```
// base.h
#ifndef BASIC_H
#define BASIC_H

class HalloWeltApp : public wxApp {
   public: virtual bool OnInit();
};

class BasicFrame : public wxFrame {
public:
   BasicFrame( const wxChar *title,
               int xpos, int ypos,
               int width, int height);
   ~BasicFrame();
};
#endif
```

In dieser Header-Datei finden Sie folgende zwei Klassen:

- `HalloWeltApp` – diese Klasse wird abgeleitet von der `wxWidgets`-Basisklasse `wxApp`. Diese Klasse enthält eine virtuelle Methode `OnInit()`.
- `BasicFrame` – diese Klasse wird von der Klasse `wxFrame` abgeleitet. Hier haben wir nur einen Konstruktor und einen Destruktor deklariert.

Die Definitionen implementieren wir jetzt in der Quelldatei *base.cpp*:

```
// base.cpp
#include <wx/wx.h>
#include "base.h"

IMPLEMENT_APP(HalloWeltApp)
```

```
bool HalloWeltApp::OnInit() {
   BasicFrame *frame =
      new BasicFrame( wxT("Hallo Welt"), 50, 50, 450, 300);
   frame->Show(TRUE);
   SetTopWindow(frame);
   return TRUE;
}

BasicFrame::BasicFrame (
   const wxChar *title,
   int xpos, int ypos,
   int width, int height)
   : wxFrame ( (wxFrame *) NULL,
               -1, title,
               wxPoint(xpos, ypos),
               wxSize(width, height)) { }

BasicFrame::~BasicFrame() { }
```

Zunächst finden Sie am Anfang dieser Datei das Makro

```
IMPLEMENT_APP( HalloWeltApp )
```

Dieses Makro konstruiert das `HalloWeltApp`-Objekt und stellt auch den `main`-Einsprungspunkt für unsere Applikation zur Verfügung. Unter MS Windows wäre dies beispielsweise die `int WINAPI WinMain()`-Funktion. Bei Linux sieht dies selbstverständlich wieder anders aus – aber im Grunde kann uns das egal sein, weil uns `wxWidgets` die plattformspezifische Arbeit abnimmt.

Die Methode `OnInit()` der Klasse `wxApp` erzeugt dann eine Instanz von `BasicFrame` mit den Werten der Position (bezogen auf die linke obere Ecke des Desktops), der Breite und Höhe des Fensters und dem Fenstertitel. Ein Blick auf den Konstruktor von `BasicFrame` zeigt, dass wir mit `BasicFrame` wiederum nur einen »Strumpf« über das eigentliche Frame `wxFrame` gezogen haben. Anschließend wird die Methode `Show()` aufgerufen, die allerdings keine Methode der `wxFrame`-Klasse ist, sondern von der Klasse `wxWindow` vererbt wurde. Damit wird das Fenster angezeigt.

[»] **Hinweis**

Das mag jetzt noch suspekt erscheinen, aber was es mit Klassen wie `wxApp`, `wxFrame` oder `wxWindow` auf sich hat, wird sich auf den nächsten Seiten klären.

Das Programm bei der Ausführung ist in Abbildung 10.2 zu sehen.

Abbildung 10.2 »Hallo Welt« mit »wxWidgets«

Hinweis [«]

In diesem Buch werden Sie immer wieder auf das Makro `wxT()` stoßen. `wxT()` ist ein Makro, das mit Zeichen- und String-Literalen verwendet (also 'c' und »xyz«) wird. Es konvertiert diese automatisch von Unicode in die eingebaute Unicode-Konfiguration. Mit diesem Makro wird praktisch der Wert eines Literals angepasst, ohne dieses in ein ASCII-Format zu ändern. Für weitere Informationen empfehle ich die Dokumentation »Unicode Overview« von `wxWidgets`.

10.3.3 Die grundlegende Struktur eines »wxWidgets«-Programms

In diesem Abschnitt soll nun etwas genauer auf den grundlegenden Aufbau einer `wxWidgets`-Applikation eingegangen werden. Darauf aufbauend verläuft die Verwendung von `wxWidgets` immer recht ähnlich.

Die Klasse `wxFrame`, die Sie im Beispiel zuvor verwendet haben, stellt uns ein Frame-Fenster zur Verfügung. Ein solches Frame ist ein einfaches Fenster, dessen Größe man ändern kann. Dieses Fenster hat einen Rand und einen Titel. Optional kann es auch Dinge wie eine Menüleiste, Statusleiste etc. enthalten. Das Frame stellt also immer einen Behälter dar, der weitere Elemente umfassen kann – selbstverständlich auch ein weiteres Frame. `wxFrame` besitzt als Basisklasse `wxWindow`.

Vielleicht haben Sie sich beim letzten Beispiel die Frage gestellt, warum wir nicht direkt die Klasse `wxFrame` verwendet und stattdessen eine neue Klasse (`BasicFrame`) erstellt und von dieser abgeleitet haben. Dies hat einen einfachen und OOP-typischen Grund: Auf diesem Weg können Sie ohne großen Aufwand eine neue Funktionalität in einer eigenen Frame-Klasse hinzufügen (oder natürlich auch nicht benötigte Elemente ignorieren).

Hierzu die Syntax des Konstruktors von wxFrame:

```
wxFrame( wxWindow* parent,
         wxWindowID id,
         const wxString& title,
         const wxPoint& pos = wxDefaultPosition,
         const wxSize& size = wxDefaultSize,
         long style = wxDEFAULT_FRAME_STYLE,
         const wxString& name = "frame"  );
```

Es folgen nun einige Parameter, die etwas näher erläutert werden sollen:

- parent – ein Zeiger auf das Eltern-Fenster. Wenn es kein Eltern-Fenster gibt, wird hierbei NULL angegeben.
- id – die Identifizierung des Fensters; –1 steht für Default.
- title – der Name des Fensters, der beim Anzeigen des Fensters in der Titelbar steht.
- pos – die Position, an der das Frame angezeigt werden soll. Geben Sie hierbei (-1,-1) an, wird die Default-Position (wxDefaultPosition) verwendet. Die Default-Position hängt von der Plattform (genauer dem Window-Manager) ab.
- size – die Größe des Fensters. Auch hier geben Sie mit (-1,-1) eine Default-Größe (wxDefaultSize) an. Und auch hier hängt die Default-Größe von der Plattform (genauer dem verwendeten Window-Manager) ab.
- style – der Frame-Stil. wxFrame bietet hierfür mehrere Styles an (einige dieser Styles finden Sie in Tabelle 10.1).
- name – der Name des Frames. Dieser Parameter wird in Verbindung mit anderen Elementen benötigt.

Style	Bedeutung
wxDEFAULT_FRAME_STYLE	eine Kombination der Stile wxMINIMIZE_BOX, wxMAXIMIZE_BOX, wxRESIZE_BOX, wxSYSTEM_MENU und wxCAPTION
wxICONIZE	Das Frame erscheint minimiert auf dem Bildschirm und besitzt keine Fensterleiste, mit der sich das Fenster minimieren, maximieren oder schließen lässt.
wxCAPTION	Das Frame besitzt zwar eine Titelleiste, aber ohne die Button zum Minimieren, Maximieren und Schließen.
wxMINIMIZE	wie wxICONIZE
wxMINIMIZE_BOX	Eine minimierbare Box wird angezeigt (ohne Fensterleiste.)
wxMAXIMIZE	Das Frame wird maximiert angezeigt (ohne Fensterleiste). Damit können Sie praktisch den kompletten Desktop abdecken.

Tabelle 10.1 Verschiedene Frame-Stile

Style	Bedeutung
wxMAXIMIZE_BOX	Eine maximierbare Box wird angezeigt (ohne Fensterleiste).
wxSTAY_ON_TOP	Das Frame wird immer als vorderstes Frame auf dem Fenster angezeigt, auch wenn Sie ein anderes Fenster fokussieren (ohne Fensterleiste).
wxSYSTEM_MENU	Ein Systemmenü wird angezeigt (Rechtsklick mit der Maus, ohne Fensterleiste).
wxSIMPLE_BORDER	Es wird kein Rand angezeigt (nur für GTK und MS Windows).
wxRESIZE_BORDER	Ein veränderbarer Rand wird angezeigt (nur UNIX).
wxFRAME_FLOAT_ON_PARENT	Das Frame befindet sich immer über dem Eltern-Fenster in der z-Ordnung und wird nicht in der Taskbar angezeigt.
wxFRAME_TOOL_WINDOW	Zeigt nur eine schmale Fensterleiste an und wird nicht in der Taskbar angezeigt (funktioniert nur mit wxDEFAULT_FRAME_STYLE | wxFRAME_TOOL_WINDOW).

Tabelle 10.1 Verschiedene Frame-Stile (Forts.)

Hinweis [«]

Wenn man mit diesen Stilen experimentiert, merkt man, dass ein Frame kein Fenster im eigentlichen Sinne ist und eben von wxWindow abgeleitet wird. Einige Stile sind außerdem plattformabhängig – wird eine Angabe nicht unterstützt, wird dies ignoriert. Sollten Sie nicht wissen, wie das Fenster zu schließen ist, hilft die Tastenkombination Alt + F4 weiter. Außerdem sieht man bei einigen dieser Stile erst einen Effekt, wenn man diese mit dem bitweisen ODER und beispielsweise mit wxDEFAULT_FRAME_STYLE verknüpft.

Elemente zum Frame hinzufügen

Ohne jetzt auf die Details der einzelnen Elemente einzugehen, sollen nun Elemente zum Frame hinzugefügt werden. Das Prinzip ist eigentlich simpel. Sie erzeugen ein Frame, wie schon geschehen, und packen in das Frame ein weiteres (GUI-)Element (oder mehrere Elemente). Im gleich folgenden Beispiel wird in das Frame ein wxPanel-Element gepackt. Dieses Element wird in der Praxis dazu verwendet, um andere Kontrollelemente zu arrangieren. In diesem Beispiel wollen wir nur einen einfachen Button (wxButton) hinzufügen. Würden Sie diesen Button gleich in das Frame stecken, so wäre das komplette Frame ein Button. Damit Sie die Größe und Position des Buttons anpassen können, benötigen Sie einen weiteren Behälter – was hier wxPanel wäre (siehe auch Abbildung 10.3).

Hinweis [«]

In diesem Abschnitt geht es noch nicht um die Details der einzelnen Kontrollelemente und von deren Parameter, sondern nur darum, wie eine typische wxWidgets-Anwendung erstellt wird.

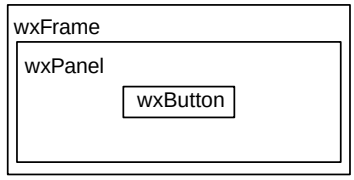

Abbildung 10.3 Ein »wxFrame« enthält weitere Elemente

Somit müssen Sie in der Klasse des Frames (hier `BasicFrame`) erst mal in der Header-Datei *base.h* die neuen Elemente hinzufügen:

```
// base.h
#ifndef BASIC_H
#define BASIC_H

class HalloWeltApp : public wxApp {
   public: virtual bool OnInit();
};

class BasicFrame : public wxFrame {
private:
   wxPanel* panel;
   wxButton* button;
public:
   BasicFrame( const wxChar *title,
               int xpos, int ypos,
               int width, int height);
   ~BasicFrame();
};
#endif
```

Das Einbauen dieser beiden Elemente (`wxPanel` und `wxButton`) in das Frame ist auch nicht viel schwerer. Hierzu müssen Sie nur noch mit `new` Speicherplatz für die einzelnen Elemente reservieren. Dazu rufen wir gleich bei der Speicherreservierung den entsprechenden Konstruktor auf und initialisieren die Elemente mit den entsprechenden Werten. Hier die Quelldatei *base.cpp*:

```
// base.cpp
#include <wx/wx.h>
#include "base.h"

IMPLEMENT_APP(HalloWeltApp)

bool HalloWeltApp::OnInit() {
   BasicFrame *frame =
```

```
      new BasicFrame( wxT("Hallo Welt"), 50, 50, 200, 100);
   frame->Show(TRUE);
   SetTopWindow(frame);
   return TRUE;
}

BasicFrame::BasicFrame (
   const wxChar *title,
   int xpos, int ypos,
   int width, int height)
   : wxFrame ( (wxFrame *) NULL,
               -1, title,
               wxPoint(xpos, ypos),
               wxSize(width, height),wxDEFAULT_FRAME_STYLE) {
   // Ein neues Panel zum Frame hinzufügen
   panel = new wxPanel( this, wxID_ANY,
       wxDefaultPosition, wxSize(200, 100) );
   // Einen Button in den Panel hinzufügen
   button = new wxButton( panel, wxID_OK,
            wxT("OK"), wxPoint(10, 10), wxDefaultSize );
}

BasicFrame::~BasicFrame() { }
```

Sicherlich stellen Sie sich jetzt die Frage, warum wir beim Code den Speicher für `wxPanel` und `wxButton` nicht mehr freigeben. Dies ist nicht nötig, weil `BasicFrame` als Elternteil den Speicher für alle Kinder freigibt, wenn dieser zerstört wird.

Das Programm bei der Ausführung:

Abbildung 10.4 Ein »wxFrame« mit einem »wxPanel« und »wxButton«

Ereignisse behandeln

Wenn Sie das Panel und den Button in den Frame eingefügt haben, benötigen Sie noch etwas, mit dem Sie auf das Drücken des Buttons reagieren können. Für solche Fälle müssen Sie die Ereignisse gesondert behandeln. Solche Ereignisse werden in `wxWidgets` über Callback-Funktionen abgefangen und behandelt. Hier soll

nur kurz darauf eingegangen werden. Mehr zur Behandlung von Ereignissen finden Sie im nächsten Abschnitt 10.3.4, »Event-Handle (Ereignisse behandeln)«.

Jedes Fenster, in dem Ereignisse behandelt werden müssen, benötigt eine Ereignis-Tabelle. Hierzu wird das Makro `DECLARE_EVENT_TABLE` verwendet. Außerdem benötigen Sie zusätzlich noch eine Methode, um auf ein Ereignis entsprechend zu reagieren. Das Ereignis, das von einem Button zurückgegeben wird, ist `wxCommandEvent`. Das ist auch gleichzeitig der Typ des Parameters der Methode, die beim Auftreten des Ereignisses ausgeführt wird. Hier zunächst wieder die Header-Datei *base.h*, in der Sie das Makro `DECLARE_EVENT_TABLE` und die Deklaration der Methode wiederfinden:

```
// base.h
#ifndef BASIC_H
#define BASIC_H

class HalloWeltApp : public wxApp {
   public: virtual bool OnInit();
};

class BasicFrame : public wxFrame {
private:
   wxPanel* panel;
   wxButton* button;
   // Ereignis-Tabelle einrichten
   DECLARE_EVENT_TABLE()
public:
   BasicFrame( const wxChar *title,
               int xpos, int ypos,
               int width, int height);
   ~BasicFrame();
   // Methode, die auf das Ereignis reagiert
   void OnClickButtonOK(wxCommandEvent &event);
};
#endif
```

Die Ereignis-Tabelle selbst wird im Quelltext *base.cpp* implementiert. Auch hier hilft uns `wxWidgets` wieder mit Makros weiter. Zunächst benötigen Sie das Makro `BEGIN_EVENT_TABLE`, mit dem die Ereignis-Tabelle beginnt. Da Sie in einer Anwendung mehrere Ereignis-Tabellen einrichten können, müssen Sie als Parameter den Namen der Klasse angeben, für die das Eintreffen eines Ereignisses von Bedeutung ist. Das Ereignis unseres Buttons selbst fügen wir mit dem Makro `EVT_BUTTON` in der Ereignis-Tabelle ein. Das Ende der Ereignis-Tabelle wird mit dem Makro `END_EVENT_TABLE` angegeben. Hierzu nun der komplette Quelltext von *base.cpp*:

```
// base.cpp
#include <wx/wx.h>
#include "base.h"

IMPLEMENT_APP(HalloWeltApp)

bool HalloWeltApp::OnInit() {
   BasicFrame *frame =
      new BasicFrame( wxT("Hallo Welt"), 50, 50, 200, 100);
   frame->Show(TRUE);
   SetTopWindow(frame);
   return TRUE;
}

BasicFrame::BasicFrame (
   const wxChar *title,
   int xpos, int ypos,
   int width, int height)
   : wxFrame ( (wxFrame *) NULL,
               -1, title,
               wxPoint(xpos, ypos),
               wxSize(width, height),wxDEFAULT_FRAME_STYLE) {
   // Ein neues Panel zum Frame hinzufügen
   panel = new wxPanel( this, wxID_ANY,
       wxDefaultPosition, wxSize(200, 100) );
   // Einen Button in den Panel hinzufügen
   button = new wxButton( panel, wxID_OK, wxT("OK"),
       wxPoint(10, 10), wxDefaultSize );
}

BasicFrame::~BasicFrame() { }

BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   EVT_BUTTON(wxID_OK, BasicFrame::OnClickButtonOK)
END_EVENT_TABLE()

void BasicFrame::OnClickButtonOK(wxCommandEvent &event ) {
   // Fenster schließen/beenden
   Close();
}
```

Wenn Sie das Programm nun ausführen und auf den Button OK klicken, wird das Fenster geschlossen und die Anwendung beendet.

10.3.4 Event-Handle (Ereignisse behandeln)

Bevor auf die gewöhnlichen Dialoge und Elemente von `wxWidgets` eingegangen werden kann, müssen zunächst noch die Grundlagen zu den *Ereignissen* behandelt werden. Denn ohne Behandlung der Ereignisse nützt Ihnen eine GUI-Applikation recht wenig.

Zu den Ereignissen zählen viele alltägliche Dinge, die man als normaler User täglich durchführt, ohne sich wahrscheinlich Gedanken darüber zu machen, wie so etwas programmtechnisch realisiert wird. Wenn Sie zum Beispiel auf Ihrem Desktop mit dem Mauszeiger auf ein Fenster gehen, die Scrollbar bewegen, Menüs anwählen, ein Fenster vergrößern, verkleinern oder minimieren, treten immer bestimmte Ereignisse auf. Auf viele solcher »Kleinigkeiten« zu reagieren, erscheint zunächst relativ komplex.

Allerdings ist das Thema, auf Ereignisse zu reagieren, im Grunde nicht `wxWidgets`-spezifisch, sondern typisch für alle GUI-Applikationen. Die Behandlung von Ereignissen läuft immer recht ähnlich ab. Die Anwendung befindet sich in einer Dauerschleife und wartet auf das Eintreten eines Ereignisses. Dies kann ein einfacher Tasten- oder Mausdruck sein. Tritt ein Ereignis ein, wird nachgesehen, ob für dieses Ereignis ein Handle eingerichtet wurde. Wenn dies der Fall ist, wird der Handle aufgerufen und ausgeführt. Wurde für ein bestimmtes Ereignis kein Handle eingerichtet, wird die Anwendung ohne Zwischenfälle weiter ausgeführt.

Der einzige Unterschied zwischen den verschiedenen Frameworks liegt in der Art und Weise, wie ein solcher Ereignis-Handle verwendet wird. `wxWidgets` verwendet hierfür eine Ereignis-Tabelle.

Ohne zu sehr auf die Details einzugehen, sind folgende Dinge nötig, um eine statische Ereignis-Tabelle zu erzeugen:

- Zunächst muss eine neue Klasse deklariert werden, die entweder direkt oder indirekt von der Klasse `wxEvtHandle` abgeleitet wurde. `wxEvtHandle` ist also die absolute Basisklasse der Ereignisse.
- Als Nächstes benötigen Sie eine Methode, die ausgeführt werden soll, wenn ein bestimmtes Ereignis aufgetreten ist.
- In der Klasse, in der eine Ereignis-Tabelle verwendet wird, müssen Sie diese mit dem Makro `DECLARE_EVENT_TABLE` deklarieren.
- Im Quellcode müssen Sie dann die Ereignis-Tabelle mit `BEGIN_EVENT_TABLE` ... `END_EVENT_TABLE` implementieren.
- In dieser Ereignis-Tabelle müssen Sie jetzt die einzelnen Einträge hinzufügen. Hierzu wird ein bestimmtes Ereignis mit einer bestimmten Klassenmethode (oder auch Ereignis-Handle-Methode) verknüpft. Alle diese Ereignisklassen-

Methoden haben als Rückgabewert den Typ `void`, sind niemals virtuell (`virtual`) und haben immer ein Ereignis-Objekt als Argument. Der Typ des Arguments hängt wiederum vom Typ des Ereignisses ab, das behandelt werden soll. Bei einfachen Menüs oder Buttons wäre dies die `wxCommandEvent`-Klasse. Wird die Größe des Fensters verändert, tritt ein Ereignis der Klasse `wxSizeEvent` auf. Bei einfachen Fällen (wie einem Knopfdruck) kann man das Ereignis-Objekt sogar ignorieren.

Wir wollen unser Beispiel aus dem letzten Abschnitt um weitere Ereignis-Handles erweitern. Dem Button haben wir nun ein Menü und eine Statusleise (unten) hinzugefügt. Wird in diesem Beispiel im Menü INFO • ABOUT angewählt, soll auch dieses Ereignis abgefangen werden. Daraufhin soll ein Text in der Statusleiste ausgegeben werden. Wir wollen hier auch das Verändern der Fenstergröße behandeln. Hierzu folgt nun also die erweiterte Header-Datei *base.h*.

Hinweis **[«]**

Auch hier werden wir noch nicht auf Einzelheiten eingehen, sondern lediglich die Ereignisse behandeln. Einzelheiten wie Menüs, Buttons usw. werden in den folgenden Abschnitten noch näher beschrieben.

```
// base.h
#ifndef BASIC_H
#define BASIC_H

class HalloWeltApp : public wxApp {
   public: virtual bool OnInit();
};

class BasicFrame : public wxFrame {
private:
   wxPanel* panel;
   wxButton* button;
   wxMenuBar *MenuBar;
   wxMenu *InfoMenu;
   enum {
     MENU_INFO_ABOUT
   };
   // Ereignis-Tabelle einrichten
   DECLARE_EVENT_TABLE()
public:
   BasicFrame( const wxChar *title,
               int xpos, int ypos,
               int width, int height);
   ~BasicFrame();
```

```
   // Methoden, die auf Ereignisse reagieren
   void OnClickButtonOK(wxCommandEvent &event);
   void OnMenuInfoAbout(wxCommandEvent &event);
   void OnSize(wxSizeEvent& event);
};
#endif
```

Hierzu auch noch die Definitionen im Quellcode *base.cpp* mit einer anschließenden Erläuterung:

```
// base.cpp
#include <wx/wx.h>
#include "base.h"

IMPLEMENT_APP(HalloWeltApp)

bool HalloWeltApp::OnInit() {
   BasicFrame *frame =
      new BasicFrame( wxT("Ereignisse behandeln"),
                      50, 50, 300, 200 );
   frame->Show(TRUE);
   SetTopWindow(frame);
   return TRUE;
}

BasicFrame::BasicFrame (
   const wxChar *title,
   int xpos, int ypos,
   int width, int height)
   : wxFrame ( (wxFrame *) NULL,
               -1, title,
               wxPoint(xpos, ypos),
               wxSize(width, height),wxDEFAULT_FRAME_STYLE) {
   // Ein neues Panel zum Frame hinzufügen
   panel = new wxPanel(this, wxID_ANY,
       wxDefaultPosition, wxSize(200, 100));
   // Einen Button in den Panel hinzufügen
   button = new wxButton(
     panel, wxID_OK, wxT("Ende"),
     wxPoint(10,10), wxDefaultSize);

   // Eine Menübar erzeugen
   MenuBar = new wxMenuBar();
   // Ein Menü erzeugen
   InfoMenu = new wxMenu();
```

```
   InfoMenu->Append(MENU_INFO_ABOUT, wxT("&About") );
   MenuBar->Append( InfoMenu, wxT("&Info") );
   SetMenuBar(MenuBar);
   // Eine einfache Statusbar erstellen
   CreateStatusBar(1);
}

BasicFrame::~BasicFrame() { }

// Ereignis-Tabelle
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   EVT_BUTTON(wxID_OK, BasicFrame::OnClickButtonOK)
   EVT_MENU(MENU_INFO_ABOUT, BasicFrame::OnMenuInfoAbout)
   EVT_SIZE(BasicFrame::OnSize)
END_EVENT_TABLE()

// Ereignis-Handles, die aufgerufen werden, wenn ...

// ... der OK-Button betätigt wurde
void BasicFrame::OnClickButtonOK(wxCommandEvent &event ) {
   Close();
}

// ... wenn im Menü "Info" ausgewählt wurde
void BasicFrame::OnMenuInfoAbout(wxCommandEvent &event) {
   SetStatusText(wxT("Hier könnte eine Info stehen ..."));
}

// ... wenn die Fenstergröße verändert wurde
void BasicFrame::OnSize(wxSizeEvent& event) {
   SetStatusText(wxT("Fenstergrösse wurde verändert!"));
}
```

Das Programm bei der Ausführung:

Abbildung 10.5 Behandeln von Ereignissen

Zunächst erfolgt ein Blick auf die Ereignis-Tabelle, die dem Frame erlaubt, auf Menü-, Button- und Größenveränderungs-Ereignisse zu reagieren.

```
// Ereignis-Tabelle für BasicFrame
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   EVT_BUTTON( wxID_OK, BasicFrame::OnClickButtonOK )
   EVT_MENU( MENU_INFO_ABOUT, BasicFrame::OnMenuInfoAbout )
   EVT_SIZE( BasicFrame::OnSize )
END_EVENT_TABLE()
```

Wenn der Anwender in diesem Beispiel auf das Info-Menüelement klickt und das Menü About auswählt, wird ein Ereignis an das Frame gesendet. Die Ereignis-Tabelle von `BasicFrame` teilt dann `wxWidgets` mit, dass ein Ereignis mit der Identifikation `MENU_INFO_ABOUT` aufgetreten ist, und ruft (falls vorhanden) den entsprechenden Handle `BasicFrame::OnMenuInfoAbout` auf.

Dass dies auch funktioniert, ist dem `EVT_`-Makro (hier im Menü: `EVT_MENU`) zu verdanken, mit dem die Ereignis-Handle-Methode `OnMenuInfoAbout` aufgerufen wird, wenn das Menüelement Info innerhalb der Hierarchie von `BasicFrame` ausgewählt wurde, das die Identifikation `MENU_INFO_ABOUT` besitzt.

Das Makro `EVT_SIZE` hingegen benötigt keine Identifikation als Argument, weil dieses Ereignis nur von dem Objekt behandelt werden kann, das es generiert hat.

Der Eintrag `EVT_BUTTON` ruft die Ereignis-Handle-Methode `OnClickButtonOK` auf, wenn innerhalb der Hierarchie von `BasicFrame` ein Button mit der Identifikation `wxID_OK` gedrückt wurde.

Das mag etwas undurchschaubar erscheinen, daher soll der Ablauf noch genauer erläutert werden. Wurde beispielsweise im Menü Info ausgewählt, überprüft `wxWidgets` erst einmal die Klasse `wxMenu` (mit den Unterklassen) nach einem passenden Ereignis-Handle. Wenn keiner gefunden wurde, wird in der darüberliegenden Klasse danach gesucht. In unserem Beispiel wäre dies die `BasicFrame`-Instanz – und hier wird mit `BasicFrame::OnMenuInfoAbout` ein entsprechender Ereignis-Handle in der Ereignis-Tabelle gefunden.

[»] **Hinweis**

Vielleicht finden Sie das Wort »Ereignis-Handle-Methode« etwas seltsam. »Ereignis-Handle« könnte aber darüber hinwegtäuschen, dass es sich hierbei auch um eine Methode handelt, und »Methode« könnte den Eindruck vermitteln, dass es sich um eine ganz normale Methode handelt.

Ein Event überspringen

Wenn ein Ereignis eintrifft, ruft `wxWidgets` die Methode `wxEvtHandle::ProcessEvent` auf. `ProcessEvent` wiederum sucht dann nach dem erstbesten Handle für das Ereignis (vereinfacht ausgedrückt). Dieser Vorgang wurde bereits im letzten Abschnitt beschrieben. Doch dieser an sich normale Vorgang kann auch ein Problem mit sich bringen. Wenn Sie wollen, dass eine direkte oder indirekte Basisklasse das Ereignis erhalten soll und nicht Ihre von der Basisklasse abgeleitete Klasse, geht dies nicht ohne weiteres. Wollen Sie dennoch, dass eine der Basisklassen das Ereignis erhält, können Sie im Ereignis-Handle das Ereignis mit der Methode `wxEvent::Skip` überspringen, so dass erneut `ProcessEvent` aufgerufen und nach einem passenden Handle in der Basisklasse weitergesucht wird. Falls kein Handle gefunden wird, wird das Programm ohne irgendwelche Zwischenfälle fortgeführt.

Dynamischer Event-Handle

Neben der gezeigten Möglichkeit, Ereignis-Handles statisch zu verwenden, können Sie diese auch dynamisch bei der Applikations-Klasse mit der Methode `OnInit()` hinzufügen – wir sprechen hier von einer dynamischen Ereignis-Tabelle.

Hierfür gibt es zwei Methoden, die mit einer dynamischen Ereignis-Tabelle arbeiten: `wxEvtHandle::Connect` und `wxEvtHandle::Disconnect`. Einen Aufruf von `wxEvtHandle::Disconnect` kann man sich aber sparen, weil eine Trennung von selbst stattfindet, wenn das Fenster-Objekt zerstört wird.

Um einen dynamischen Ereignis-Handle zu verwenden, benötigen Sie die Makros `DECLARE_EVENT_TABLE` in *base.h* und alles von `BEGIN_EVENT_TABLE` bis `END_EVENT_TABLE` in *base.cpp* nicht mehr. Sie müssen in der Header-Datei *base.h* nur noch die Methode deklarieren, die beim Eintreten eines Ereignisses aufgerufen wird. Außerdem haben wir uns beim Listing jetzt auf ein Menüelement beschränkt:

```
// base.h
#ifndef BASIC_H
#define BASIC_H

class HalloWeltApp : public wxApp {
   public: virtual bool OnInit();
};

class BasicFrame : public wxFrame {
private:
   wxMenuBar *MenuBar;
   wxMenu *InfoMenu;
```

```
public:
   BasicFrame( const wxChar *title,
               int xpos, int ypos,
               int width, int height);
   ~BasicFrame();

   // Methoden, die auf Ereignisse reagieren
   void OnMenuInfoAbout(wxCommandEvent &event);
};
#endif
```

In der Quelldatei *base.cpp* müssen Sie nur noch ein paar Zeilen in der Methode `OnInit()` der Applikations-Klasse einfügen (und natürlich weiterhin die Methode (Handle) definieren, die beim Eintreten des Ereignisses aufgerufen werden soll):

```
// base.cpp
#include <wx/wx.h>
#include "base.h"

IMPLEMENT_APP(HalloWeltApp)

bool HalloWeltApp::OnInit() {
   BasicFrame *frame =
      new BasicFrame( wxT("Ereignisse behandeln"),
                      50, 50, 300, 200);
   frame->Show(TRUE);
   // Dynamischer Ereignis-Handle
   frame->Connect(
      wxID_ABOUT, wxEVT_COMMAND_MENU_SELECTED,
      wxCommandEventHandler(BasicFrame::OnMenuInfoAbout ) );
   SetTopWindow(frame);
   return TRUE;
}

BasicFrame::BasicFrame (
   const wxChar *title,
   int xpos, int ypos,
   int width, int height)
   : wxFrame ( (wxFrame *) NULL,
               -1, title,
               wxPoint(xpos, ypos),
               wxSize(width, height), wxDEFAULT_FRAME_STYLE)
{
   // Eine Menübar erzeugen
```

```
    MenuBar = new wxMenuBar();
    // Ein Menü erzeugen
    InfoMenu = new wxMenu();
    InfoMenu->Append(wxID_ABOUT, wxT("&About"));
    MenuBar->Append( InfoMenu, wxT("&Info"));
    SetMenuBar(MenuBar);
    // Eine einfache Statusbar erstellen
    CreateStatusBar(1);
}

BasicFrame::~BasicFrame() { }

// ... wenn im Menü "Info" ausgewählt wurde
void BasicFrame::OnMenuInfoAbout(wxCommandEvent &event) {
    SetStatusText(wxT("Hier könnte eine Info stehen ..."));
}
```

Der Methode `Connect` übergeben wir als ersten Parameter die Fenster-Identifikation (`wxID_ABOUT`), dann den Ereignis-Identifizierer (`wxEVT_COMMAND_MENU_SELECTED`) und zuletzt einen Zeiger auf die Ereignis-Handle-Methode. Beachten Sie, dass der Ereignis-Identifizierer mit `wxEVT_COMMAND_MENU_SELECTED` anders lautet als beim Makro der Ereignis-Tabelle (`EVT_MENU`). Dieses andere Makro ist nötig, weil sonst versucht würde, eine statische Ereignis-Tabelle zu erzeugen.

Weiteres zu den Event-Handles

Natürlich ließe sich zu den Ereignissen noch eine ganze Menge mehr schreiben. Aber dieses Kapitel soll nur eine Einführung in die `wxWidgets`-Programmierung bieten. Nicht behandelt wurde die Möglichkeit, die Funktion eines Ereignis-Handles zu verändern bzw. zu erweitern und eigene Ereignisse zu definieren. Hierzu sei bei Bedarf ein Blick in die Dokumentation von `wxWidgets` empfohlen (siehe auch Buch-CD).

10.3.5 Die Fenster-Grundlagen

Hinweis **[«]**

Der Ausdruck *Fenster* gefällt mir zwar nicht, aber wenn ich hier von Window-Grundlagen rede, hört sich das genauso unpassend und ein wenig nach Microsoft an. Hier haben es die englischschreibenden Autoren einfacher.

Jeder wird zwar wissen, was ein Fenster ist, aber ein Programmierer, der die `wxWidgets`-API verwenden will, muss sich intensiver mit dem Modell eines Fensters befassen, das `wxWidgets` verwendet. Anschließend können wir uns dann mit den einzelnen Elementen eines Fensters befassen.

Im Grunde ist ein Fenster ein rechteckiger Bereich mit einigen eingestellten Eigenschaften wie »Fenstergröße verändern«, »verstecken«, »anzeigen« usw. Ein Fenster kann selbstverständlich auch weitere Fenster enthalten. Ein Fenster einer `wxWidgets`-Anwendung ist gewöhnlich ein Objekt der Klasse `wxWindow` bzw. (meistens) einer davon abgeleiteten Klasse.

Des Weiteren unterscheidet man zwischen einem Client- und einem Nicht-Client-Bereich. Wenn Sie ein gewöhnliches Fenster vor sich haben, können Sie die Größe des Fensters ändern, die Dekoration, den Rahmen und den Titel. Dabei handelt es sich um den Nicht-Client-Bereich. Zum Client-Bereich gehört alles, was sich innerhalb des Fensters befindet, wie zum Beispiel eine Menübar, die Statusleiste etc.; auch weitere Fenster (sogenannte *Kind-Fenster*) können hierbei verwendet werden.

Top-Level-Fenster

Fenster werden aufgeteilt in *Top-Level-Fenster* (`wxFrame`, `wxDialog`, `wxPopup`) und andere Fenster. Top-Level-Fenster werden immer ohne ein Eltern-Fenster (`NULL`) erzeugt. Abgesehen von `wxPopup` haben alle Top-Level-Fenster gewöhnlich eine Dekoration wie den Titel, die Knöpfe zum Schließen, Minimieren, Maximieren und zum Ändern der Größe und noch einiges mehr. Natürlich können diese Angaben verändert werden.

Koordinatensystem

Beim Erzeugen eines Fensters wird auch ein Koordinatensystem verwendet, um zu bestimmen, wo das Fenster auf dem Bildschirm erscheinen und wie groß es sein soll. Unabhängig davon, welche Angaben Sie hierbei machen, ist der Punkt (0, 0) immer die linke obere Ecke. Zur Angabe der Größe und Position wird das Pixel verwendet.

Fenster erzeugen und zerstören

Gewöhnlich wird ein neues Fenster mit `new` auf dem Heap erzeugt. Bei den meisten Fenstern ist dazu nicht mehr als ein Schritt nötig. Bei einigen Fenstern, ausgenommen `wxFrame` und `wxDialog`, müssen Sie beim ersten Parameter das Eltern-Fenster mit angeben und können hierbei nicht `NULL` beim Konstruktor verwenden. Sobald ein Fenster zerstört wird, werden automatisch auch alle darin befindlichen Elemente – auch weitere Fenster – zerstört.

Wenn Sie ein Fenster oder Top-Level-Fenster erzeugen, wird dies standardmäßig immer angezeigt. Wollen Sie ein Fenster erzeugen, das zunächst nicht sichtbar erzeugt wird (minimiert), müssen Sie nur die Methode `Show(false)` aufrufen. `Show` ist standardmäßig auf `true` gesetzt.

Um ein Fenster zu zerstören, wird die Methode `Destroy` für Top-Level-Fenster (oder `delete` für Kind-Fenster) aufgerufen. Hierbei wird das Ereignis `wxEVT_DESTROY` gesendet, bevor das Fenster zerstört wird.

Hinweis **[«]**

Neben der gängigen Vorgehensweise, ein Fenster mit `new` und dem Konstruktor zu erstellen, gibt es für alle Fenster auch noch die Möglichkeit, sie in zwei Schritten zu erzeugen. Man verwendet dazu den Standardkonstruktor und ruft anschließend die Methode `Create`() mit den entsprechenden Parametern auf. Jede Fensterklasse besitzt diese `Create`-Methode, worauf allerdings in diesem Buch nicht näher eingegangen wird.

10.3.6 Übersicht über die »wxWidgets«-(Fenster-)Klassen

Bevor es um die Interna eines Fensters geht, soll hier noch ein kurzer Überblick über die üblichen Fensterklassen gegeben werden, die man in fast allen Anwendungen findet. Natürlich gibt es auch noch viele weitere Fensterklassen, aber wir begnügen uns zunächst mit den Basics.

- **Basisfensterklassen** – in der `wxWidgets`-Anwendung werden diese Basisklassen gewöhnlich nie direkt verwendet. Darin enthalten ist in der Regel die grundlegende Funktionalität für die abgeleiteten Klassen.
  - `wxWindow` – die Basisklasse für alle (Top-Level-)Fenster
  - `wxControl` – die Basisklasse für alle einfachen Kontrollelemente wie beispielsweise `wxButton`
  - `wxControlWithItems` – die Basisklasse für alle Kontrollelemente mit mehreren Elementen (wie beispielsweise `wxListBox` und `wxCheckListBox`)
- **Top-Level-Fenster** – das sind gewöhnlich die Fenster, die am meisten verwendet und unverzüglich nach dem Erzeugen auf dem Bildschirm angezeigt werden.
  - `wxFrame` – ein in der Größe veränderbares Fenster, das auch andere Fenster enthält
  - `wxMDIParentFrame` – ein Frame, das andere Frames verwaltet
  - `wxMDIChildFrame` – ein Frame, das vom Eltern-Frame verwaltet wird
  - `wxDialog` – ein in der Größe veränderbares Fenster um Abfragen durchzuführen (z. B. die beliebte »Sind Sie sicher?«-Abfrage)
  - `wxPopupWindow` – ein schlichtes Fenster ohne besondere Ausstattung

- **Container-Fenster** – diese Fenster sind Behälter, die Kind-Fenster verwalten.
  - `wxPanel` – ein Container-Fenster, mit dem Sie das Layout kontrollieren können
  - `wxNotebook` – ein Container-Fenster, um zwischen den einzelnen Seiten mit Tabs hin und her zu schalten
  - `wxScrolledWindow` – ein Container-Fenster zum Scrollen der Unterfenster
  - `wxSplitterWindow` – ein Container-Fenster, das zwei Unterfenster verwaltet
- **Nicht statische Kontrollelemente** – diese Elemente können vom Anwender verändert werden.
  - `wxButton` – ein gewöhnlicher Button (Knopf) mit einem Text
  - `wxBitmapButton` – ein gewöhnlicher Button (Knopf) mit einem Bitmap
  - `wxChoice` – eine Dropdown-Liste zum Auswählen
  - `wxComboBox` – ein editierbares Feld mit einer Liste zum Auswählen
  - `wxCheckBox` – Checkboxen (Häkchen an, Häkchen aus)
  - `wxListBox` – eine Liste mit auswählbaren Strings
  - `wxRadioBox` – ähnlich wie die Checkboxen, nur eben als Punkt
  - `wxScrollBar` – eine Leiste zum Scrollen
  - `wxSpinButton` – Pfeil, an dem ein Wert inkrementiert/dekrementiert wird
  - `wxSpinCtrl` – ein Textfeld und Spin-Button, um einen Integer zu modifizieren
  - `wxTextCtrl` – ein einfaches oder mehrzeiliges Texteingabefeld
  - `wxToggleButton` – ein Button, den man aktivieren oder deaktivieren kann
- **Statische Kontrollelemente** – dies sind Elemente, die Informationen anzeigen und vom Anwender nicht verändert werden können.
  - `wxGauge` – eine Anzeige, wie weit ein Vorgang schon abgeschlossen ist
  - `wxStaticText` – ein Text-Label
  - `wxStaticBitmap` – ein Bitmap-Label
  - `wxStaticLine` – eine Linie
  - `wxStaticBox` – eine Box, die um statische und nicht statische Kontrollelemente herumgelegt werden kann
- **Menüs** – Menüs enthalten eine Liste von Kommandos.
  - `wxMenu` – ein Menü kann als Popup oder in einer Menübar verwendet werden.

- **Kontrollbar** – eine Kontrollbar enthält gewöhnlich Kommandos und Informationen.
  - `wxMenuBar` – eine Menübar, die Kommandos in einem `wxFrame` enthält
  - `wxToolBar` – mit einer Toolbar haben Sie einen schnelleren Zugriff auf Kommandos.
  - `wxStatusBar` – eine Statusbar zeigt Informationen an (gegebenenfalls in mehreren Feldern).

10.3.7 »wxWindow«, »wxControl« und »wxControlWithItems« – die Basisklassen

`wxWindow` ist die Basisklasse für alle Fenster und alle sichtbaren Objekte auf dem Bildschirm. Alle Kontrollelemente, Top-Level-Fenster usw. sind direkt oder indirekt von `wxWindow` abgeleitet. Dies bedeutet natürlich auch, dass `wxWindow` das erste Element ist, das erzeugt, und das letzte, das zerstört wird.

In der Praxis werden Sie wohl kaum die Basisklasse `wxWindow` direkt verwenden – aber diese Klasse enthält viele wichtige Methoden und Ereignisse, die in den abgeleiteten Klassen ebenfalls zur Verfügung stehen. Diese Methoden zu beschreiben würde bedeuten, nur die Basisklasse `wxWindow` zu beschreiben, was Einsteiger nur verwirren würde. Auf einige dieser Methoden wird bei Bedarf in den davon abgeleiteten Klassen eingegangen. Wenn Sie weitere Details erfahren möchten, empfehle ich Ihnen die API-Referenz von `wxWidgets`.

Im Folgenden werden die Ereignisse beschrieben, die bei `wxWindow` und davon abgeleiteten Klassen generiert werden können, und es wird erläutert, wie Sie den entsprechenden Handle (mit dem entsprechenden Makro) einrichten können (siehe Tabelle 10.2). Wie Sie einen Ereignis-Handle einrichten können, haben Sie bereits in Abschnitt 10.3.4, »Event-Handle (Ereignisse behandeln)«, erfahren.

Handle einrichten	Beschreibung
`EVT_WINDOW_CREATE (func)`	Wird durch `wxEVT_CREATE` aufgerufen und generiert, wenn ein Fenster gerade erzeugt wurde. Der Handle benötigt als Parameter ein Objekt vom Typ `wxWindowCreateEvent`.
`EVT_WINDOW_DESTROY (func)`	Wird durch `wxEVT_DELETE` aufgerufen und generiert, wenn das Fenster zerstört (geschlossen) wird. Der Handle benötigt als Parameter ein Objekt vom Typ `wxWindowDestroyEvent`.
`EVT_PAINT (func)`	Wird durch `wxEVT_PAINT` aufgerufen und generiert, wenn das Fenster erneuert werden muss. Der Handle benötigt als Parameter ein Objekt vom Typ `wxPaintEvent`.

Tabelle 10.2 »wxWindow«-Ereignisse

Handle einrichten	Beschreibung
EVT_ERASE_BACKGROUND (func)	Wird durch wxEVT_ERASE_BACKGROUND aufgerufen und generiert, wenn der Fensterhintergrund erneuert werden muss. Der Handle benötigt als Parameter ein Objekt vom Typ wxEraseEvent.
EVT_MOVE (func)	Wird durch wxEVT_MOVE aufgerufen und generiert, wenn das Fenster verschoben wurde. Der Handle benötigt als Parameter ein Objekt vom Typ wxMoveEvent.
EVT_SIZE (func)	Wird durch wxEVT_SIZE aufgerufen und generiert, wenn die Größe des Fensters verändert wurde. Der Handle benötigt als Parameter ein Objekt vom Typ wxSizeEvent.
EVT_SET_FOCUS (func) EVT_KILL_FOCUS (func)	Wird durch wxEVT_SET_FOCUS und wxEVT_KILL_FOCUS aufgerufen und generiert, wenn die Tastatur oder die Maus den Fokus auf das Fenster erhält oder verliert. Der Handle benötigt als Parameter ein Objekt vom Typ wxFocusEvent.
EVT_SYS_COLOUR_CHANGED (func)	Wird durch wxEVT_SYS_COLOUR_CHANGED aufgerufen und generiert, wenn der Anwender die Farben des Kontroll-Panels (wxPanel) verändert hat (nur MS Windows). Der Handle benötigt als Parameter ein Objekt vom Typ wxSysColourChangedEvent.
EVT_IDLE (func)	Wird durch wxEVT_IDLE aufgerufen und im Leerlauf der Anwendung generiert. Der Handle benötigt als Parameter ein Objekt vom Typ wxIdleEvent.
EVT_UPDATE_UI (func)	Wird durch wxEVT_UPDATE_UI aufgerufen und generiert, wenn die Leerlaufzeit dem Fenster die Möglichkeit gibt, sich zu erneuern.

Tabelle 10.2 »wxWindow«-Ereignisse (Forts.)

Hierzu folgt nun ein einfaches Beispiel, wie Sie diese Ereignisse in der Praxis behandeln können. Im Beispiel wurden Ereignis-Handles für das Verschieben und das Verändern der Größe eingerichtet. Das aktuelle Ereignis wird in der Statuszeile angezeigt. Zunächst die Header-Datei *base.h*:

```
// base.h
#ifndef BASIC_H
#define BASIC_H

class WindowEventApp : public wxApp {
   public: virtual bool OnInit();
};

class BasicFrame : public wxFrame {
private:
   // Ereignis-Tabelle einrichten
   DECLARE_EVENT_TABLE()
```

```
public:
   BasicFrame( const wxChar *title,
               int xpos, int ypos,
               int width, int height);
   ~BasicFrame();

   // Methoden, die auf Ereignisse reagieren
   void OnSize(wxSizeEvent& event);
   void OnMove(wxMoveEvent& event);
};
#endif
```

Jetzt noch die Quelldatei *base.cpp*:

```
// base.cpp
#include <wx/wx.h>
#include "base.h"

IMPLEMENT_APP(WindowEventApp)

bool WindowEventApp::OnInit() {
   BasicFrame *frame =
      new BasicFrame( wxT("wxWindow-Ereignisse"),
                      50,50,300,200 );
   frame->Show(TRUE);
   SetTopWindow(frame);
   return TRUE;
}

BasicFrame::BasicFrame (
   const wxChar *title,
   int xpos, int ypos,
   int width, int height)
   : wxFrame ( (wxFrame *) NULL,
               -1, title,
               wxPoint(xpos, ypos),
               wxSize(width, height), wxDEFAULT_FRAME_STYLE)
{
   // Eine einfache Statusbar erstellen
   CreateStatusBar(1);
}

BasicFrame::~BasicFrame() { }

// Ereignis-Tabelle
```

```
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   EVT_SIZE(BasicFrame::OnSize)
   EVT_MOVE(BasicFrame::OnMove)
END_EVENT_TABLE()

// Ereignis-Handles, die aufgerufen werden, wenn

// ... die Fenstergröße verändert wurde
void BasicFrame::OnSize(wxSizeEvent& event) {
   SetStatusText(wxT("Fenstergrösse wurde verändert!"));
}

// ... das Fenster verschoben wird
void BasicFrame::OnMove(wxMoveEvent& event) {
   SetStatusText(wxT("Fenster wurde verschoben!"));
}
```

»wxControl« und »wxControlWithItems«

`wxControl` wurde von der Klasse `wxWindow` abgeleitet und ist die abstrakte Basisklasse für alle Kontrollelemente, bei denen der Anwender eine Eingabe machen kann. Dies sind zum Beispiel Fenster, die Daten anzeigen, oder Elemente, bei denen der Anwender mit der Maus oder Tastatur Kommandos senden kann.

`wxControlWithItems` ist ebenfalls eine abstrakte Basisklasse, die von `wxWindow` abgeleitet wurde. Diese Klasse wird auch für Kontrollelemente verwendet, die mehrere Elemente enthalten (beispielsweise `wxLixtBox`, `wxCheckListBox`, `wxChoice` oder `wxComboBox`).

Auch die beiden abstrakten Basisklassen enthalten viele Methoden, die wiederum von den davon abgeleiteten Klassen verwendet werden.

10.3.8 Top-Level-Fenster

Die Top-Level-Fenster werden im Gegensatz zu `wxWindow` immer direkt auf dem Bildschirm platziert. Ein Top-Level-Fenster kann natürlich auch andere Fenster enthalten, die Größe kann verändert werden, wenn die Anwendung es erlaubt, etc.

Es gibt drei grundlegende Top-Level-Fenster. `wxFrame` und `wxDialog` sind abgeleitet von der abstrakten Basisklasse `wxTopLevelWindow`. Das dritte Top-Level-Fenster, `wxPopupWindow`, ist ein Fenster mit eingeschränkter Funktionalität. `wxPopupWindow` wurde direkt von `wxWindow` abgeleitet.

Ein Fenster mit `wxDialog` kann so erzeugt werden, dass die Anwendung, die das Dialogfenster aufgerufen hat, angehalten wird, bis der Anwender diesen Dialog schließt. Es kann aber auch ein `wxDialog`-Fenster erzeugt werden, das die Anwendung nicht anhält.

Gewöhnlich haben Top-Level-Fenster eine Titelbar und als zusätzliche Ausstattung kleine Knöpfe für das Minimieren, Maximieren und Schließen des Fensters. Ein Frame kann zusätzlich noch eine Menüleiste, eine Werkzeugleiste und eine Statusleiste enthalten, was ein Dialogfenster gewöhnlich nicht hat.

wxFrame

Zwar wurde `wxFrame` bereits kurz behandelt, da dieses Top-Level-Fenster aber in einer `wxWidgets`-Anwendung am häufigsten verwendet wird, soll hier etwas genauer darauf eingegangen werden.

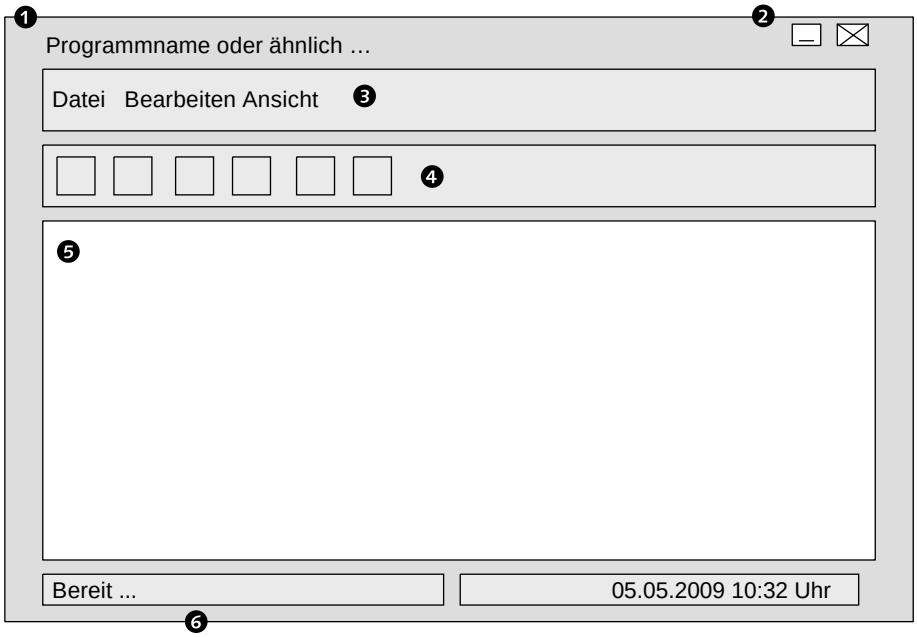

Abbildung 10.6 Die Elemente eines Frames

In Abbildung 10.6 können Sie die grundlegenden Elemente eines Frames sehen, die Sie von den meisten Anwendungen, die sich auf Ihrem Rechner befinden, kennen sollten. Dies sind (bezogen auf die Nummern in Abbildung 10.6): ❶ Titelleiste, ❷ Fensterrahmen und Ausstattung, ❸ Menüleiste (`wxMenuBar`), ❹ Werkzeugleiste (`wxToolBar`), ❺ Client-Bereich, ❻ Statusleiste (`wxStatusBar`).

Um ein Frame zu zerstören, sollten Sie die Methoden `Destroy` oder `Close` statt `delete` verwenden. Sobald dann alle Ereignisse abgearbeitet wurden, wird auch das Frame zerstört. An das Fenster wird hierbei das Ereignis `wxEVT_CLOSE` gesendet. Sofern Sie keinen eigenen Ereignis-Handle für dieses Ereignis eingerichtet haben, ruft der Standard-Ereignis-Handle dann `Destroy` auf. Wenn ein Frame gelöscht wird, werden automatisch auch alle darin enthaltenen Elemente gelöscht.

`wxFrame` und die davon abgeleiteten Klassen haben dieselben Ereignisse wie `wxWindow` (siehe Tabelle 10.2). Zusätzlich finden Sie in `wxFrame` aber noch weitere Ereignisse (siehe Tabelle 10.3).

Handle einrichten	Beschreibung
`EVT_ACTIVATE(func)`	Wird von `wxEVT_ACTIVATE` aufgerufen und generiert, wenn ein Frame aktiviert oder deaktiviert wird. Der Handle benötigt als Parameter ein Objekt vom Typ `wxActivateEvent`.
`EVT_CLOSE(func)`	Wird von `wxEVT_CLOSE` aufgerufen und generiert, wenn das Programm oder das Fenstersystem versucht, diesen Frame zu schließen. Der Handle benötigt als Parameter ein Objekt vom Typ `wxCloseEvent`.
`EVT_ICONIZE(func)`	Wird von `wxEVT_ICONIZE` aufgerufen und generiert, wenn der Frame minimiert oder wiederhergestellt wird. Der Handle benötigt als Parameter ein Objekt vom Typ `wxIconizeEvent`.
`EVT_MAXIMIZE(func)`	Wird von `wxEVT_MAXIMIZE` aufgerufen und generiert, wenn ein Frame maximiert oder wiederhergestellt wurde. Der Handle benötigt als Parameter ein Objekt vom Typ `wxMaximizeEvent`.

Tabelle 10.3 »wxFrame«-Ereignisse

`wxFrame` verfügt neben den eigenen Methoden auch über die Methoden der Basisklassen `wxTopLevelWindow` und `wxWindow`. Daher sollten Sie bei Bedarf wieder einen Blick in die Dokumentation werfen. Auf alle Methoden von `wxFrame` wird hier nicht sofort eingegangen. Im Verlauf des Kapitels werden Sie aber noch einige Methoden von `wxFrame` kennenlernen.

Methode	Beschreibung
`virtual wxStatusBar* CreateStatusBar( int number = 1, long style = 0, wxWindowID id = -1, const wxString& name = "statusBar" );`	Erzeugt unten am Frame eine Statusleiste. Mit `number` geben Sie an, in wie viele Felder diese Leiste eingeteilt werden soll (Standard ist 1). Auch den Stil können Sie mit `style` angeben (siehe `wxStatusBar`). Die Identifikation wird mit `id` und der Name mit `name` angegeben.

Tabelle 10.4 Einige Methoden von »wxFrame«

Methode	Beschreibung
`virtual void SetStatusText( const wxString& text, int number = 0);`	Setzt einen Text in die Statusleiste und zeichnet diese dann neu. Mit `text` geben Sie den String ein und mit `number` das Statusleistenfeld (angefangen bei 0), in dem der Text erscheinen soll. Wenn Sie eine leere Statusleiste wollen, verwenden Sie einen leeren String.
`void Centre( int direction = wxBOTH );`	Zentriert ein Frame auf dem Bildschirm. Mögliche Angaben für `direction` sind: `wxHORIZONTAL`, `wxVERTICAL` oder `wxBOTH`.

Tabelle 10.4 Einige Methoden von »wxFrame« (Forts.)

Interessant ist hierbei auch die Methode `wxFrame::CreateToolbar`, mit der eine Werkzeugleiste unter der Menüleiste erzeugt und mit dem Frame verbunden wird. Alternativ hierzu können Sie auch die Klasse `wxToolBar` verwenden. Um aber eine Werkzeugleiste mit dem Frame zu verwalten, benötigen Sie trotzdem die Methode `wxFrame::SetToolBar`. Mehr dazu erfahren Sie bei der Beschreibung der Klasse `wxToolBar`.

Auch für die Menüleiste finden Sie mit `wxFrame::GetMenuBar` und `wxFrame::SetMenuBar` zwei Vermittler für die Frames und die Klasse `wxMenuBar`. Weitere Informationen dazu finden Sie weiter unten bei der Beschreibung der Klasse `wxMenuBar`.

Methoden aus der Klasse `wxTopLevelWindow`, die bei den Frames häufig zum Einsatz kommen, sind folgende:

Methode	Beschreibung
`wxString GetTitle() const;`	Holt sich den String des Fenstertitels.
`virtual void SetTitle( const wxString& title );`	Setzt den String des Fenstertitels auf `title`.
`void Iconize(bool iconize);`	Iconisiert das Fenster (so dass es in der Taskleiste des Bildschirms erscheint). Wenn hier `true` angegeben wird, wird das Fenster iconisiert und mit `false` wiederhergestellt. Ob das Fenster bereits iconisiert ist, können Sie mit `IsIconize()` abfragen.
`void Maximize(bool maximize);`	Maximiert das Fenster. Wird als Parameter `true` angegeben, wird ein Fenster maximiert. `false` stellt den alten Zustand wieder her. Ob das Fenster bereits maximiert ist, können Sie mit `IsMaximize()` abfragen.

Tabelle 10.5 Einige Methoden von »wxTopLevelWindow«

Methode	Beschreibung

`bool ShowFullScreen( bool show, long style = wxFULLSCREEN_ ALL);`	Abhängig vom Wert `style` kann das Fenster im Vollbildschirm angezeigt werden (und überdeckt somit alles). Folgende Stile können Sie hierbei für `style` verwenden:
	`wxFULLSCREEN_NOMENUBAR`
	`wxFULLSCREEN_NOTOOLBAR`
	`wxFULLSCREEN_NOSTATUSBAR`
	`wxFULLSCREEN_NOBORDER`

	`wxFULLSCREEN_NOCAPTION` `wxFULLSCREEN_ALL` (alles zusammen) Mit `true` als erstem Parameter geben Sie an, dass zum Vollbildmodus geschaltet werden soll, und mit `false` stellen Sie den ursprünglichen Modus wieder her. Wollen Sie testen, ob sich das Fenster bereits im Vollbildmodus befindet, können Sie dies mit `IsFullScreen()` machen.

Tabelle 10.5 Einige Methoden von »wxTopLevelWindow« (Forts.)

Und natürlich kann man sich zusätzlich noch die Methoden von `wxWindow` aneignen.

[»] Hinweis

Um sich eingehend mit `wxWidgets` zu befassen, sollten Sie immer auch die Dokumentation – besonders die Referenz – zu Rate ziehen. Neben den Methoden von `wxFrame` können Sie zum Beispiel auch auf die Methoden von `wxTopLevelWindow` und `wxWindow` zugreifen. Dabei kommen an die 100 Methoden (!) zusammen.

Natürlich soll auch wieder ein kleines Beispiel erstellt werden, das die Statusleiste, das Maximieren und den Vollbildschirm demonstriert. Um die einzelnen Methoden auszuführen, wurde ein Menü verwendet (auf die Menüs wird in Abschnitt 10.3.12, »Menüs«, noch eingegangen). Hier zunächst die Header-Datei *base.h*:

```
// base.h
#ifndef BASIC_H
#define BASIC_H

class wxFrameDemoApp : public wxApp {
   public: virtual bool OnInit();
};

class BasicFrame : public wxFrame {
private:
   wxMenuBar *MenuBar;
```

```
   wxMenu    *ExampleMenu;
   enum {
     MENU_EXAMPLE_MAXIMIZE,
     MENU_EXAMPLE_CENTRE,
     MENU_EXAMPLE_FULLSCREEN
   };
   // Ereignis-Tabelle einrichten
   DECLARE_EVENT_TABLE()
public:
   BasicFrame( const wxChar *title,
               int xpos, int ypos,
               int width, int height);
   ~BasicFrame();

  // Methoden, die auf Ereignisse reagieren
  void OnMenuExampleMaximize( wxCommandEvent &event);
  void OnMenuExampleMaximizeStatusBar( wxMenuEvent& event );
  void OnMenuExampleCentre( wxCommandEvent &event);
  void OnMenuExampleCentreStatusBar( wxMenuEvent& event );
  void OnMenuExampleFullScreen( wxCommandEvent &event);
  void OnMenuExampleFullScreenStatusBar(wxMenuEvent& event);
};
#endif
```

Jetzt noch die Quelldatei *base.cpp*:

```
// base.cpp
#include <wx/wx.h>
#include "base.h"

IMPLEMENT_APP(wxFrameDemoApp)

bool wxFrameDemoApp::OnInit() {
   BasicFrame *frame =
      new BasicFrame( wxT("Demonstriert wxFrame"),
                      50, 50, 500, 300);
   frame->Show(TRUE);
   SetTopWindow(frame);
   return TRUE;
}

BasicFrame::BasicFrame (
   const wxChar *title,
   int xpos, int ypos,
   int width, int height)
```

```
   : wxFrame ( (wxFrame *) NULL,
               -1, title,
               wxPoint(xpos, ypos),
               wxSize(width, height),wxDEFAULT_FRAME_STYLE)
{
   // Eine Menübar erzeugen
   MenuBar = new wxMenuBar();
   // Ein Menü erzeugen
   ExampleMenu = new wxMenu();
   ExampleMenu->Append(
      MENU_EXAMPLE_MAXIMIZE, wxT("&Maximize" ));
   ExampleMenu->Append(MENU_EXAMPLE_CENTRE, wxT("&Centre"));
   ExampleMenu->Append(
      MENU_EXAMPLE_FULLSCREEN, wxT("&ShowFullScreen"));
   MenuBar->Append( ExampleMenu, wxT("&Example"));
   SetMenuBar(MenuBar);
   // Eine einfache Statusbar mit zwei Spalten erstellen
   CreateStatusBar(2);
}

BasicFrame::~BasicFrame() { }

// Ereignis-Tabelle
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   EVT_MENU(MENU_EXAMPLE_MAXIMIZE,
      BasicFrame::OnMenuExampleMaximize)
   EVT_MENU_HIGHLIGHT( MENU_EXAMPLE_MAXIMIZE,
      BasicFrame::OnMenuExampleMaximizeStatusBar )
   EVT_MENU(MENU_EXAMPLE_CENTRE,
      BasicFrame::OnMenuExampleCentre)
   EVT_MENU_HIGHLIGHT( MENU_EXAMPLE_CENTRE,
      BasicFrame::OnMenuExampleCentreStatusBar )
   EVT_MENU(MENU_EXAMPLE_FULLSCREEN,
      BasicFrame::OnMenuExampleFullScreen)
   EVT_MENU_HIGHLIGHT( MENU_EXAMPLE_FULLSCREEN,
      BasicFrame::OnMenuExampleFullScreenStatusBar )
END_EVENT_TABLE()

// Ereignis-Handles, die aufgerufen werden, wenn ...

// ... im Menü "Info" ausgewählt wurde
void BasicFrame::OnMenuExampleMaximize(
   wxCommandEvent &event) {
   if(IsMaximized()) {
      SetStatusText(wxT("Maximale Größe erreicht ...") ,0);
```

```
      SetStatusText(wxT(""),1);
   }
   else {
      Maximize();
      SetStatusText(wxT(""),1);
   }
}

// ... der Mauszeiger sich auf der
// Menüauswahl "Maximize" befindet.
void BasicFrame::OnMenuExampleMaximizeStatusBar(
   wxMenuEvent& event ) {
   SetStatusText(wxT("Fenster maximieren ...?"),1);
   SetStatusText(wxT(""),0);
}

// ... im Menü "Centre" ausgewählt wurde
void BasicFrame::OnMenuExampleCentre(wxCommandEvent &event){
   Centre();
   SetStatusText(wxT("Fenster zentriert ..."), 0);
   SetStatusText(wxT(""),1);
}

// ... Info beim Mouseover auf der Menüauswahl "Centre"
void BasicFrame::OnMenuExampleCentreStatusBar(
      wxMenuEvent& event ) {
   SetStatusText(wxT("Fenster zentrieren ...?"), 1);
   SetStatusText(wxT(""), 0);
}

// ... im Menü "Centre" ausgewählt wurde
void BasicFrame::OnMenuExampleFullScreen(
   wxCommandEvent &event) {
   if(IsFullScreen()) {
       // Bereits im FullScreen-Modus
   }
   else {
      ShowFullScreen(TRUE);
      SetStatusText(wxT(""),1);
   }
}

// ... sich der Mauszeiger auf der
// Menüauswahl "Centre" befindet.
void BasicFrame::OnMenuExampleFullScreenStatusBar(
```

```
    wxMenuEvent& event ) {
    SetStatusText(
        wxT("Auf Vollbildschirm umschalten...?") ,1 );
    SetStatusText(wxT("") ,0);
}
```

Das Programm bei der Ausführung:

Abbildung 10.7 Einige Methoden von »wxFrame« bei der Ausführung

»wxMDIParentFrame« und »wxMDIChildFrame«

Die beiden Klassen `wxMDIParentFrame` und `wxMDIChildFrame` sind abgeleitet von der Klasse `wxFrame` und ein Teil von `wxWidgets`-*Multi-Document-Interface*-Fenstern (kurz *MDI-Fenster*). Damit wird erreicht, dass ein Eltern-Fenster mehrere `wxMDIChildFrame`-Fenster verwaltet. In Abbildung 10.8 sehen Sie ein Beispiel, das im Allgemeinen der wxWidgets-Bibliothek als Listing beiliegen sollte.

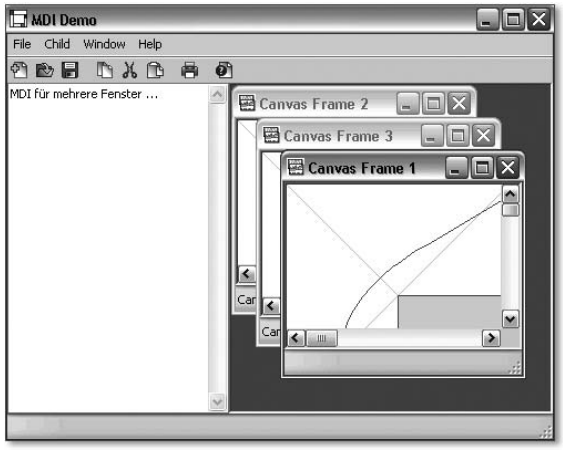

Abbildung 10.8 Multi Document Interface (MDI)

`wxMDIChildFrame` wird normalerweise als Kind eines `wxMDIParentFrame` erzeugt. Mehr dazu entnehmen Sie bitte der Dokumentation zu `wxWidgets`.

»wxDialog« (mit »wxMessageDialog«)

Eine Dialogbox ist ein Fenster mit einer Titelleiste und neben `wxFrame` eine weitere häufig verwendete Fensterklasse. Dieses Fenster kann verschoben werden und die Kontrolle aller anderen Fenster abschalten. Man spricht hierbei vom *Modal-* und *Modeless-*Modus. Eine Modal-Dialogbox blockiert den Fluss des Programms, bis die Dialogbox geschlossen wird.

Hinweis **[«]**

Bitte verwechseln Sie diese Dialogbox nicht mit der typischen Nachrichtenbox (`wxMessageDialog`) – die allerdings auch davon abgeleitet ist.

Der Konstruktor von `wxDialog` hat folgende Syntax:

```
wxDialog( wxWindow* parent,
          wxWindowID id,
          const wxString& title,
          const wxPoint& pos = wxDefaultPosition,
          const wxSize& size = wxDefaultSize,
          long style = wxDEFAULT_DIALOG_STYLE,
          const wxString& name = "dialogBox" );
```

Die einzelnen Parameter haben die folgende Bedeutung:

- `parent` – das Eltern-Fenster, das diesen Dialog erhält. Gibt es kein Eltern-Fenster, wird `NULL` angegeben. Ansonsten kann das Eltern-Fenster ein Frame oder eine andere Dialogbox sein.
- `id` – die Identifizierung des Fensters
- `title` – der Fenstertitel des Dialogs
- `pos` – die Position des Dialogfensters. Geben Sie hierbei (-1,-1) an, wird die Default-Position (`wxDefaultPosition`) verwendet. Die Default-Position hängt von der Plattform (genauer vom Window-Manager) ab.
- `size` – die Größe des Dialogfensters. Auch hier geben Sie mit (-1,-1) eine Default-Größe (`wxDefaultSize`) an. Und auch hier hängt die Default-Größe von der Plattform (genauer dem verwendeten Window-Manager) ab.
- `style` – der Fensterstil des Dialogfensters (siehe Tabelle 10.6)
- `name` – der Name für die Kontrolle

wxDialog (style)	Beschreibung
wxCAPTION	Fügt die Überschrift mit Titelleiste in die Dialogbox ein.
wxDEFAULT_DIALOG_STYLE	Der Vorgabewert; entspricht einer Kombination aus wxCAPTION, wxCLOSE_BOX und wxSYSTEM_MENU.
wxRESIZE_BORDER	Die Dialogbox kann in der Größe verändert werden.
wxSYSTEM_MENU	Zeigt ein Systemmenü beim Rechtsklick mit der Maus am oberen Fensterrahmen an (nicht unter UNIX).
wxCLOSE_BOX	Zeigt die Box zum Schließen an.
wxMAXIMIZE_BOX	Zeigt die Box zum Maximieren an.
wxMINIMIZE_BOX	Zeigt die Box zum Minimieren an.
wxTHICK_FRAME	Zeigt einen dicken Rahmen um das Fenster an.
wxSTAY_ON_TOP	Zeigt die Dialogbox vor allen anderen Fenstern an.
wxNO_3D	Es werden keine 3D-Rahmen verwendet (nur MS Windows).

Tabelle 10.6 Stile von »wxDialog«

Wenn Sie Dialoge in den Programmen verwenden, benötigen Sie eine Ressourcen-Datei. Wenn Sie hierbei keine speziellen Icons brauchen, kann diese Ressourcen-Datei mit der Endung *.rc* folgendermaßen aussehen:

```
#include "wx/msw/wx.rc"
```

Neben den Methoden von wxTopLevelWindow und wxWindow bietet auch wxDialog einige Methoden an (siehe Tabelle 10.7). Die wichtigsten werden hier kurz erwähnt – ansonsten sei, wie immer, ein Blick in die Dokumentation empfohlen.

Methode	Beschreibung
`void SetModal(const bool flag);`	Erlaubt dem Programmierer einzustellen, ob die Dialogbox modal (wxDialog::Show blockiert andere Fenster so lange, bis die Dialogbox geschlossen wird) oder modeless (die Kontrolle der anderen Fenster bleibt weiterhin vorhanden) sein soll. Mit true stellen Sie den Dialog auf »modal« und mit false auf »modeless«.
`int ShowModal();`	Zeigt einen modalen Dialog an. Der Programmfluss kehrt erst dann zurück, wenn der Dialog geschlossen oder mit wxDialog::EndModal aufgehoben wurde. Den Rückgabewert können Sie zum Auswerten verwenden.

Tabelle 10.7 Einige Methoden von »wxDialog«

Das folgende Beispiel zeigt, wie Sie eine Dialogbox erstellen können, die eine typische About-Information anzeigt. Die Header-Datei *base.h* dazu:

```
// base.h
#ifndef BASIC_H
#define BASIC_H

class wxDialogDemoApp : public wxApp {
   public: virtual bool OnInit();
};

class BasicFrame : public wxFrame {
private:
   wxMenuBar *MenuBar;
   wxMenu    *ExampleMenu;
   enum {
     MENU_INFO_ABOUT
   };
   // Ereignis-Tabelle einrichten
   DECLARE_EVENT_TABLE()
public:
   BasicFrame( const wxChar *title,
               int xpos, int ypos,
               int width, int height);
   ~BasicFrame();
   // Ereignis-Handle
   void OnMenuInfoAbout(wxCommandEvent &event);
};

class AboutDialog : public wxDialog {
   private:
   wxStaticText *m_pInfoText;
   wxButton *m_pOkButton;
   // Ereignis-Tabelle einrichten
   DECLARE_EVENT_TABLE()

   public:
   AboutDialog(wxWindow *parent);
   virtual ~AboutDialog() { }
   // Ereignis-Handle
   void SetText(const wxString& text);
   void OnClose(wxCloseEvent& event);
};
#endif
```

Die Quelldatei *base.cpp*:

```
// base.cpp
#include <wx/wx.h>
#include "base.h"
```

```
IMPLEMENT_APP(wxDialogDemoApp)

bool wxDialogDemoApp::OnInit() {
   BasicFrame *frame =
      new BasicFrame( wxT("Demonstriert wxDialog"),
                      50, 50, 300, 200);
   frame->Show(TRUE);
   SetTopWindow(frame);
   return TRUE;
}

BasicFrame::BasicFrame (
   const wxChar *title,
   int xpos, int ypos,
   int width, int height)
   : wxFrame ( (wxFrame *) NULL,
               -1, title,
               wxPoint(xpos, ypos),
               wxSize(width, height),wxDEFAULT_FRAME_STYLE)
{
   // Eine Menübar erzeugen
   MenuBar = new wxMenuBar();
   // Ein Menü erzeugen
   ExampleMenu = new wxMenu();
   ExampleMenu->Append(MENU_INFO_ABOUT, wxT("&About"));
   MenuBar->Append( ExampleMenu, wxT("&Info"));
   SetMenuBar(MenuBar);
}

BasicFrame::~BasicFrame() { }

// Ereignis-Tabelle für das Frame
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   EVT_MENU(MENU_INFO_ABOUT, BasicFrame::OnMenuInfoAbout)
   //EVT_CLOSE(BasicFrame::OnClose)
END_EVENT_TABLE()

// Ereignis-Tabelle für den Dialog
BEGIN_EVENT_TABLE(AboutDialog, wxDialog)
   EVT_CLOSE(AboutDialog::OnClose)
END_EVENT_TABLE()

void BasicFrame::OnMenuInfoAbout(wxCommandEvent &event) {
   AboutDialog *dlg = new AboutDialog(this);
   dlg->SetText(wxT("(c)2009 P.R.O.N.I.X\nJ.Wolf & Co.\n"));
   dlg->ShowModal();
}
```

```
AboutDialog:: AboutDialog(wxWindow *parent)
    : wxDialog( parent, -1, wxT("About ..."),
                wxDefaultPosition, wxSize(200, 200),
                wxDEFAULT_DIALOG_STYLE)
{
   // Einen Text für das Label
   m_pInfoText = new wxStaticText(
      this, -1, wxT(""), wxPoint(5, 5),
      wxSize(100, 100), wxALIGN_CENTRE );
   // Einen Button zum Drücken
   m_pOkButton = new wxButton(
      this, wxID_OK, wxT("Ok"), wxPoint(5, 40));
}

void AboutDialog::SetText(const wxString& text) {
   m_pInfoText->SetLabel(text);
}

void AboutDialog::OnClose(wxCloseEvent& event ) {
   Destroy();
}
```

Das Programm bei der Ausführung (im Menü auf INFO • ABOUT klicken):

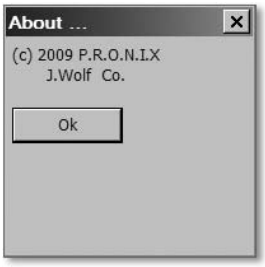

Abbildung 10.9 »wxDialog« bei der Ausführung

Im Beispiel wurde außerdem demonstriert, wie man einen zusätzlichen Ereignis-Handle für ein weiteres Fenster einrichten kann. In diesem Beispiel aber wäre der Ereignis-Handle `AboutDialog::OnClose` nicht nötig gewesen, weil Sie einen Standard-Ereignis-Handle benutzt haben. Eine Dialogbox wird hier automatisch geschlossen, wenn der Anwender auf einen `wxID_OK`- oder `wxID_CANCEL`-Button als Identifizierer drückt.

Wenn Sie nicht den Standard-Ereignis-Handle verwenden, also keinen OK-Button mit `wxID_OK` oder `wxID_CANCEL` einsetzen, müssen Sie die Dialogbox selbst zerstören, indem Sie einen Ereignis-Handle mit dem Makro `EVT_CLOSE` einrich-

ten und darin die Methode `Destroy()` aufrufen. Sollten Sie dies versäumen, wird Ihre Anwendung weiterhin im Hintergrund laufen, auch wenn Sie, wie im Beispiel, das Frame längst geschlossen haben.

Beim Ausführen des Beispiels kann man allerdings auch erkennen, dass das Layout der Dialogbox zu wünschen übriglässt. Bei einem solch einfachen Fall greift man in der Praxis aber normalerweise zur Klasse `wxMessageDialog` (eine abgeleitete Klasse von `wxDialog`).

Hierzu ein kurzer Umweg zur Klasse `wxMessageDialog`. Die Syntax des Konstruktors ist wie folgt definiert:

```
wxMessageDialog( wxWindow* parent,
                 const wxString& message,
                 const wxString& caption = "Message box",
                 long style = wxOK | wxCANCEL,
                 const wxPoint& pos = wxDefaultPosition );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster, das diesen Dialog erhält. Gibt es kein Eltern-Fenster, wird `NULL` angegeben. Ansonsten kann das Eltern-Fenster ein Frame oder eine andere Dialogbox sein.
- `message` – die Nachricht, die in der Nachrichtenbox angezeigt wird
- `caption` – die Überschrift der Nachrichtenbox
- `style` – hier werden die Flags für die einzelnen Buttons und Icons angegeben (mögliche Werte finden Sie in Tabelle 10.8).
- `pos` – die Position des Dialogs (nicht des Fensters)

Stil	Beschreibung
`wxOK`	ein OK-Button
`wxCANCEL`	ein Cancel-Button (Abbrechen)
`wxYES_NO`	Ja- und Nein-Button
`wxYES_DEFAULT`	Ja- und Nein-Button, wobei der Ja -Button als Default vorgewählt ist
`wxNO_DEFAULT`	Ja- und Nein-Button, wobei der Nein-Button als Default vorgewählt ist
`wxICON_EXCLAMATION`	Zeigt ein Ausrufezeichen als Icon (Icon plattformabhängig).
`wxICON_HAND`	Zeigt ein Fehler-Icon an (Icon plattformabhängig).
`wxICON_ERROR`	wie `wxICON_HAND`

Tabelle 10.8 Mögliche Angaben der verschiedenen Stile (»wxMessageDialog«)

Stil	Beschreibung
wxICON_QUESTION	Zeigt ein Fragezeichen als Icon (Icon plattformabhängig).
wxICON_INFORMATION	Zeigt ein Informations-Icon (Icon plattformabhängig).
wxSTAY_ON_TOP	Die Nachrichtenbox steht immer über allen anderen Fenstern und Applikationen ganz vorn (nur MS Windows).

Tabelle 10.8 Mögliche Angaben der verschiedenen Stile (»wxMessageDialog«) (Forts.)

Zum Anzeigen einer Nachrichtenbox wird die Methode `wxMessageDialog::ShowModal` verwendet. Hierzu folgt nun ein Beispiel, das identisch mit der Ausführung im letzten Listing ist, nur dass jetzt statt `wxDialog` `wxMessageDialog` verwendet wird. Außerdem wird in diesem Beispiel demonstriert, wie man auf das Schließen einer Anwendung bzw. auf das Ereignis `wxEVT_CLOSE` reagieren kann. Zuerst wieder die Header-Datei *base.h*:

```
// base.h
#ifndef BASIC_H
#define BASIC_H

class wxDialogDemoApp : public wxApp {
   public: virtual bool OnInit();
};

class BasicFrame : public wxFrame {
private:
   wxMenuBar *MenuBar;
   wxMenu    *ExampleMenu;
   enum {
     MENU_INFO_ABOUT
   };
   // Ereignis-Tabelle einrichten
   DECLARE_EVENT_TABLE()
public:
   BasicFrame( const wxChar *title,
               int xpos, int ypos,
               int width, int height);
   ~BasicFrame();

   // Methoden, die auf Ereignisse reagieren
   void OnMenuInfoAbout(wxCommandEvent &event);
   void OnClose(wxCloseEvent& event);
};
#endif
```

Jetzt noch die Quelldatei *base.cpp*:

```
// base.cpp
#include <wx/wx.h>
#include "base.h"

IMPLEMENT_APP(wxDialogDemoApp)

bool wxDialogDemoApp::OnInit() {
   BasicFrame *frame =
      new BasicFrame( wxT("Demonstriert wxMessageDialog"),
                      50, 50, 300, 200);
   frame->Show(TRUE);
   SetTopWindow(frame);
   return TRUE;
}

BasicFrame::BasicFrame (
   const wxChar *title,
   int xpos, int ypos,
   int width, int height)
   : wxFrame ( (wxFrame *) NULL,
               -1, title,
               wxPoint(xpos, ypos),
               wxSize(width, height),wxDEFAULT_FRAME_STYLE)
{
   // Eine Menübar erzeugen
   MenuBar = new wxMenuBar();
   // Ein Menü erzeugen
   ExampleMenu = new wxMenu();
   ExampleMenu->Append(MENU_INFO_ABOUT, wxT("&About"));
   MenuBar->Append( ExampleMenu, wxT("&Info"));
   SetMenuBar(MenuBar);
}

BasicFrame::~BasicFrame() { }

// Ereignis-Tabelle für das Frame
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   EVT_MENU(MENU_INFO_ABOUT, BasicFrame::OnMenuInfoAbout)
   EVT_CLOSE(BasicFrame::OnClose)
END_EVENT_TABLE()

void BasicFrame::OnMenuInfoAbout(wxCommandEvent &event) {
```

```
   wxMessageDialog *dlg =
      new wxMessageDialog(
         this, wxT("(c) 2009 P.R.O.N.I.X\nJ.Wolf & Co.\n"),
         wxT("About..."), wxOK | wxICON_INFORMATION );
      dlg->ShowModal();
}

void BasicFrame::OnClose(wxCloseEvent& event ) {
   bool destroy = true;
   if( event.CanVeto()) {
      wxMessageDialog *dlg =
        new wxMessageDialog(
         this,
         wxT("Wollen Sie das Programm wirklich verlassen?"),
         wxT("Panik ...?"),
         wxYES_NO | wxNO_DEFAULT | wxICON_QUESTION );
      int result = dlg->ShowModal();
      if ( result == wxID_NO ) {
         event.Veto();
         destroy = false;
      }
   }
   if ( destroy ) {
      wxMessageDialog *dlg2 =
         new wxMessageDialog(
            this, wxT("Bitte auf OK drücken"),
             wxT("Bye ..."), wxOK | wxICON_EXCLAMATION );
      dlg2->ShowModal();
      Destroy();
   }
}
```

Die Abbildungen 10.10 bis 10.12 zeigen das Programm bei der Ausführung:

Abbildung 10.10 Bei Info About (»wxOK« | »wxICON_INFORMATION«)

Abbildung 10.11 »wxYES_NO« | »wxNO_DEFAULT« | »wxICON_QUESTION«

Abbildung 10.12 »wxOK« | »wxICON_EXCLAMATION«

Dazu muss noch kurz auf folgenden Codeabschnitt (ein wenig gekürzt) eingegangen werden:

```
void BasicFrame::OnClose(wxCloseEvent& event ) {
   bool destroy = true;
   if( event.CanVeto()) {
      wxMessageDialog *dlg =  new wxMessageDialog( ... );
      int result = dlg->ShowModal();
      if ( result == wxID_NO ) {
         event.Veto();
         destroy = false;
      }
   }
   if ( destroy ) {
      wxMessageDialog *dlg2 =  new wxMessageDialog( ... );
      dlg2->ShowModal();
      Destroy();
   }
}
```

Beim Schließen der Hauptanwendung haben wir mit `EVT_CLOSE` den Handle `BasicFrame::OnClose` eingerichtet. Wenn der Anwender versucht, das Fenster zu schließen, wird dieser Ereignis-Handle aufgerufen (gilt auch für `wxDialog` – eben für alle Top-Level-Fenster). Wenn Sie die Anwendung nicht beenden wollen, müssen Sie mit der Methode `wxCloseEvent::CanVeto` überprüfen, ob dies möglich ist. Wenn der Rückgabewert `true` ist, liegt es in Ihrer Hand, ob Sie die Anwendung weiter ausführen oder mit `Destroy()` beenden wollen. Wird `false` zurückgegeben, müssen Sie das Fenster mit `Destroy()` zerstören. Wenn Sie das

Fenster nicht zerstören wollen, können Sie die Methode `wxCloseEvent::Veto` aufrufen. Jetzt weiß das `wxWidgets`-System, dass Sie das Fenster nicht schließen wollen. Dies ist die einzige Möglichkeit, auf ein `Close` zu reagieren bzw. zu vermeiden, dass nach dem `Close` das Fenster geschlossen wird.

Zurück zu `wxDialog`: Ich denke, keiner dürfte mit dem Ergebnis des Listings mit `wxDialog` zufrieden sein. Um am Layout zu feilen, gibt es die Klassen `wxLayoutConstraints` und `wxSizer`. Allerdings wird `wxLayoutConstraints` nur noch wegen der Kompatibilität erwähnt – es wird empfohlen, für solche Aufgaben `wxSizer` zu verwenden.

Layout mit »wxSizer«

Alle *Sizers* (ich erspare mir hier eine Eindeutschung) gehen von der Klasse `wxSizer` aus und bekommen auch alle Methoden vererbt, die in `wxSizer` vorhanden sind. Abgeleitete Klassen von `wxSizer` sind `wxGridSizer`, `wxFlexGridSizer`, `wxBoxSizer`, `wxStaticBoxSizer` und `CreateButtonSizer`. Man kann sich nun die Frage stellen, warum man hierfür eigene Klassen verwendet und man nicht gleich beim Erstellen einer Dialogbox alles einrichten kann. Der Hauptgrund liegt darin, dass es so erst möglich wird, komplett plattformunabhängige Layouts zu erstellen. Wie auch schon bei `wxWindow` ist `wxSizer` nur die Basisklasse, die zwar viele Methoden mitbringt (und auch weitervererbt), aber in der Regel nie direkt verwendet wird.

Die wichtigste Methode aller Sizers ist `Add`. Mit ihr können Sie den Raum und die Größe der Elemente, die Sie hinzufügen wollen, kontrollieren. Zunächst müssen Sie mit dieser Methode alle Elemente den Sizers hinzufügen. In unserer Dialogbox wären dies beispielsweise der Knopf (Button) und das Text-Label. Hierzu die Syntax dieser Methode:

```
void Add( wxWindow* window,
          int option = 0,
          int flag   = 0,
          int border = 0,
          wxObject* userData = NULL );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `window` – das Fenster, das hinzugefügt werden soll
- `option` – wird zusammen mit `wxBoxSizer` verwendet. `0` bedeutet, dass sich die Größe des Widgets nicht in die Hauptorientierung des Sizers ändern darf. Wird `1` verwendet, dann darf das Widget in die Hauptrichtung vergrößert und verkleinert werden.

- `flag` – hier können Sie Flags angeben, die auch mit dem bitweisen ODER verknüpft werden können (die einzelnen Flags und deren Bedeutung finden Sie in den Tabellen 10.9 und 10.10).
- `border` – hier geben Sie die Breite des Rahmens an, wenn ein entsprechendes Flag gesetzt wurde.
- `userData` – damit können Sie ein zusätzliches Objekt angeben, das dem Sizer-Element hinzugefügt werden soll.

Flag (für Rahmen)	Beschreibung
`wxTOP`	Definiert oben einen Rahmen mit `border` Breite Pixel.
`wxLEFT`	Definiert links einen Rahmen mit `border` Breite Pixel.
`wxBOTTOM`	Definiert unten einen Rahmen mit `border` Breite Pixel.
`wxRIGHT`	Definiert rechts einen Rahmen mit `border` Breite Pixel.
`wxALL`	Definiert an allen Seiten einen Rahmen mit `border` Breite Pixel.

Tabelle 10.9 Flags für Rahmen

Flag	Beschreibung
`wxGROW` `wxEXPAND`	Das Element darf in der Größe verändert werden.
`wxSHAPED`	Das Element wird gleichmäßig vergrößert.
`wxALIGN_CENTER` `wxALIGN_CENTRE`	Das Element ist zentriert.
`wxALIGN_LEFT`	Das Element ist nach links ausgerichtet.
`wxALIGN_TOP`	Das Element ist nach oben ausgerichtet.
`wxALIGN_RIGHT`	Das Element ist nach rechts ausgerichtet.
`wxALIGN_CENTER_HORIZONTAL`	Das Element wird in horizontaler Ausrichtung zentriert.
`wxALIGN_CENTER_VERTICAL`	Das Element wird in vertikaler Ausrichtung zentriert.

Tabelle 10.10 Flags für die Ausrichtung des Elements im Rahmen

Neben der Möglichkeit, Elemente in einen Sizer zu packen, gibt es eine weitere Methode `Add` mit der Möglichkeit, Sizers hinzuzufügen. Die Syntax lautet:

```
void Add( wxSizer* sizer,
          int option = 0,
          int flag   = 0,
          int border = 0,
          wxObject* userData = NULL );
```

Abgesehen vom ersten Parameter entspricht diese Methode exakt derjenigen, die beim Hinzufügen von Elementen verwendet wird. Im Folgenden geht es nun um

die von wxSizer abgeleiteten Klassen, mit denen Sie ein eigenes Layout erstellen können.

wxBoxSizer

Ein wxBoxSizer wird für Layouts verwendet, bei denen Sie die Kontrolle von Spalten und Zeilen (ähnlich einer Tabellenkalkulation) vornehmen wollen. wxBoxSizer kann sich in alle Richtungen ausdehnen. Abhängig von den Flags, die Sie angeben, kann ein wxBoxSizer vertikal und/oder horizontal ausgedehnt werden. Dies müssen Sie beim Erzeugen der Box angeben. Hierzu nun ein Beispiel mit wxBoxSizer. Das komplette Listing hier abzudrucken kann ich mir sparen. Im Gegensatz zum Beispiel mit wxDialog müssen Sie nur den Konstruktor von AboutDialog in base.cpp ändern:

```
AboutDialog::AboutDialog(wxWindow *parent)
    : wxDialog( parent, -1, wxT("About ..."),
                wxDefaultPosition, wxSize(200, 150),
                wxDEFAULT_DIALOG_STYLE)
{
   // Eine Layout-Box für die Dialogbox
   wxBoxSizer *dialogSizer = new wxBoxSizer(wxVERTICAL);
   // Eine Layout-Box für den Text-Label
   wxBoxSizer *textSizer = new wxBoxSizer(wxHORIZONTAL);
   // Eine Layout-Box für den Button
   wxBoxSizer *buttonSizer = new wxBoxSizer(wxHORIZONTAL);

   m_pInfoText = new wxStaticText(
      this, -1, wxT(""), wxDefaultPosition,
      wxDefaultSize, wxALIGN_CENTER );
   // Der Text darf sich horizontal ausdehnen (2. Parameter)
   // Ein Rahmen wird in alle Richtungen mit der Stärke 10
   // um den Text gelegt
   textSizer->Add(m_pInfoText, 1, wxALL, 10);

   m_pOkButton = new wxButton(
      this, wxID_OK, wxT("Ok"), wxPoint(-1, -1) );
   // Auch der Button darf sich horizontal ausdehnen
   // Hier wird ein Rahmen mit der Stärke 20 nur rechts
   // und links vom Button gelegt
   buttonSizer->Add(m_pOkButton, 1, wxLEFT | wxRIGHT, 20);

   // Jetzt die beiden Sizer in den Sizer des Dialog-Fensters
   // hinzufügen. Diese können sich wiederum nur horizontal
   // ausdehnen.
   dialogSizer->Add(textSizer, 0, wxGROW);
```

```
    dialogSizer->Add(buttonSizer, 0, wxGROW);

    SetSizer(dialogSizer);
    SetAutoLayout(TRUE);
    Layout();
}
```

Wenn Sie das Programm jetzt ausführen, ergibt sich beim Anklicken von Info • About im Menü folgende Dialogbox (siehe Abbildung 10.13):

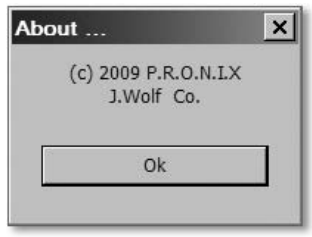

Abbildung 10.13 Das Layout von »wxDialog« mit »wxSizer« erstellt

Im Beispiel wurden außerdem auch weitere Methoden von `wxWindow` verwendet:

Methode	Beschreibung
`void SetSizer( wxSizer* sizer, bool deleteOld=true );`	Übergibt dem Fenster den Layout-Sizer. Wenn bereits ein altes Layout-Objekt existiert, wird dieses durch die Angabe des zweiten Parameters (`true`) gelöscht. Wird hierbei `false` angegeben, wird das bereits existierende Layout nicht gelöscht und weiter verwendet. Wenn der erste Parameter nicht `NULL` ist, ruft diese Methode normalerweise implizit die Methode `wxWindow::SetAutoLayout` mit `true` auf, ansonsten mit `false`.
`void SetAutoLayout( bool autoLayout );`	Legt fest, dass die Methode `wxWindow::Layout` automatisch aufgerufen wird, wenn ein Fenster in der Größe verändert wird. Normalerweise wird diese Methode von `wxWindow::SetSizer` aufgerufen; wenn Sie jedoch die Methode `wxWindow::SetConstraints` aufrufen, sollten Sie dies manuell tun.
`void Layout();`	Ruft den Layout- oder Sizer-basierenden Algorithmus für das Fenster auf.

Tabelle 10.11 Methoden von »wxWindow« für die Verwendung von »wxSizer«

wxGridSizer

Ein *Grid-Sizer* ist ein weiterer Sizer, der das Layout in einer zweidimensionalen Tabelle verwaltet. Allerdings haben dann alle Zellen dieselbe Größe. Beim Anlegen eines Objekts vom Typ `wxGridSizer` geben Sie an, wie viele Zeilen und Spal-

ten Sie verwenden wollen. Außerdem können Sie noch angeben, wie viel Platz zwischen den einzelnen Elementen frei sein soll.

Auch hierfür können wir das letzte Beispiel nutzen und müssen wiederum nur den Konstruktor anpassen. Hier der neue Konstruktor in *base.cpp*:

```
AboutDialog:: AboutDialog(wxWindow *parent)
    : wxDialog( parent, -1, wxT("About ..."),
                wxDefaultPosition, wxSize(200, 150),
                wxDEFAULT_DIALOG_STYLE)
{
   // Erzeugt einen Grid-Sizer mit zwei Zeilen und einer
   // Spalte. Der freie Raum dazwischen beträgt 10 Pixel.
   wxGridSizer *dialogSizer = new wxGridSizer(2, 1, 10, 10);

   m_pInfoText = new wxStaticText(
      this, -1, wxT(""), wxDefaultPosition,
      wxDefaultSize, wxALIGN_CENTER);
   dialogSizer->Add(m_pInfoText, 0, wxALIGN_CENTER);

   m_pOkButton = new wxButton(
      this, wxID_OK, wxT("Ok"), wxPoint(-1, -1));
   dialogSizer->Add(m_pOkButton, 0, wxALIGN_CENTER);

   SetSizer(dialogSizer);
   SetAutoLayout(TRUE);
   Layout();
}
```

Das Programm (oder besser: Dialogfenster) bei der Ausführung:

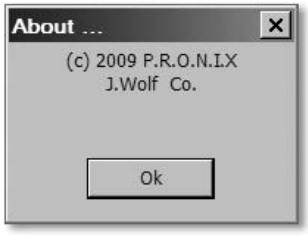

Abbildung 10.14 Ein neues Layout mit »wxGridSizer«

»wxFlexGridSizer«, »wxStaticBoxSizer« und »wxNotebookSizer«

`wxFlexGridSizer` und `wxGridSizer` sind recht ähnlich. Auch mit `wxFlexGridSizer` können Elemente in einer zweidimensionalen Tabelle angeordnet werden. Dabei haben alle Zeilen in der Reihe die gleiche Höhe und alle Zeilen in der Spalte die glei-

che Breite. wxFlexGridSizer wird gerne verwendet, wenn sich auf der einen Seite der Spalte ein Text-Label und auf der anderen Seite zum Beispiel ein Texteingabefeld befindet. Durch wxFlexGridSizer entsteht hierbei ein einheitliches Bild.

wxStaticBoxSizer ist abgeleitet von wxBoxSizer. Damit können Sie statische Boxen rund um einen Sizer hinzufügen.

WxNotebookSizer wird in Verbindung mit wxNotebook verwendet.

[»] **Hinweis**

Die Möglichkeiten, das Layout für die Anwendung zu verwalten, sind überaus vielfältig und sehr flexibel. Darauf umfassend einzugehen würde hier jedoch zu weit führen.

wxPopupWindow

wxPopupWindow ist ein weiteres Top-Level-Fenster, das eine minimale Dekoration besitzt, wie sie zum Beispiel für Tooltipps eingesetzt werden könnte. Aber da dieses Fenster nicht auf allen Systemen vorhanden ist und es für Tooltipps die Klasse wxToolTip gibt, soll auf eine Beschreibung verzichtet werden.

10.3.9 Container-Fenster

Container-Fenster werden gewöhnlich dazu verwendet, weitere visuelle Elemente in die grafische Oberfläche zu integrieren. Häufig nutzt man hierbei weitere Unterfenster oder auch grafische Zeichnungen auf dem Fenster.

»wxPanel« und »wxNotebook«

Hier werden gleich beide Container beschrieben, weil sich wxNotebook und wxPanel sehr gut eignen, GUI-Elemente für verschiedene Aufgaben in eigenen Registerkarten (Notebook) zu sammeln. Damit lassen sich die wxWidgets-GUI-Elemente sehr effizient strukturieren.

wxPanel

wxPanel ist ein vollkommenes wxWindow mit einigen dialogtypischen Eigenschaften. In der Regel wird diese Klasse verwendet, wenn Sie einige Elemente (wie zum Beispiel wxButton, wxCheckButton, wxRadioButton etc.) einsetzen wollen, ohne gleich auf das etwas unbequemere wxDialog zurückzugreifen. Am meisten werden wxPanel mit wxNotebook in der Praxis hierzu genutzt, auch Farbe lässt sich damit verwenden. Der Konstruktor der Klasse wxPanel sieht folgendermaßen aus:

```
wxPanel( wxWindow* parent,
         wxWindowID id = -1,
         const wxPoint& pos = wxDefaultPosition,
         const wxSize& size = wxDefaultSize,
```

```
        long style = wxTAB_TRAVERSAL,
        const wxString& name = "panel" );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Fenster, das diesen Dialog erhält
- `id` – die Identifizierung des Fensters. Mit `–1` geben Sie den Default-Wert an.
- `pos` – die Position des Panels; mit (–1, –1) geben Sie die Default-Position an, die vom System abhängig ist.
- `size` – die Größe des Panels; mit (–1, –1) geben Sie die Default-Größe an, die vom System abhängig ist).
- `style` – der Fensterstil, wobei `wxPanel` selbst keinen speziellen Stil definiert
- `name` – der Name für das Panel

`wxPanel` hat zwar auch einige Methoden definiert, aber diese sind hier nicht besonders erwähnenswert. Die wichtigeren Methoden sind in der Basisklasse `wxWindow` definiert. Diesbezüglich können Sie bei Bedarf einen Blick in die Dokumentation werfen. In der Praxis reicht allerdings häufig der Konstruktor aus.

wxNotebook

Mit dieser Klasse können Sie mehrere Seiten (Registerkarten) kontrollieren, indem Sie auf die einzelnen Register klicken. Eine solche Seite ist normalerweise ein `wxPanel` oder eine davon abgeleitete Klasse. Auf den meisten Systemen ist außerdem ein kleiner Button zum Scrollen dabei, wenn nicht mehr alle Register auf dem Fenster auf einmal dargestellt werden können. Dies ist leider nicht unter Mac OS der Fall – hier ist die Anzahl der Register, die angezeigt werden können, von der Größe des Fensters und der Größe des Text-Labels auf dem Register abhängig.

Zum Erzeugen eines `wxNotebook` ist folgender Konstruktor definiert:

```
wxNotebook( wxWindow* parent,
            wxWindowID id,
            const wxPoint& pos = wxDefaultPosition,
            const wxSize& size = wxDefaultSize,
            long style = 0,
            const wxString& name = wxNotebookNameStr );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster (darf nicht `NULL` sein)
- `id` – die Identifizierung des Fensters
- `pos` – die Position

- `size` – die Größe
- `style` – der Fensterstil (siehe Tabelle 10.12)
- `name` – der Name für das Panel (wird nur unter Motif benötigt)

Neben den Stilen von `wxWindow` können Sie für `wxNotebook` noch folgende Stile verwenden:

Stil	Beschreibung
`wxNB_TOP`	Platziert den Reiter des Registers oben.
`wxNB_LEFT`	Platziert den Reiter des Registers links.
`wxNB_RIGHT`	Platziert den Reiter des Registers rechts.
`wxNB_BOTTOM`	Platziert den Reiter des Registers unten.
`wxNB_FIXEDWIDTH`	Alle Register haben die gleiche Weite.
`wxNB_MULTILINE`	Wenn die Größe des Fensters nicht ausreicht, können mehrere Zeilen von Reitern erzeugt werden.
`wxNB_NOPAGETHEME`	Da es unter MS Windows auch bestimmte Themen für Notebook-Seiten gibt, können Sie dies hierbei unterbinden, was sich positiv auf die Performance des Programms auswirkt.

Tabelle 10.12 Stile für »wxNotebook«

[»] **Hinweis**

Sollte auf dem Bildschirm ein leichtes Flimmern auftreten, können Sie zusätzlich (mit dem bitweisen ODER) das Flag `wxCLIP_CHILDREN` beim Fensterstil mit angeben.

In der Klasse `wxNotebook` sind viele Methoden definiert. In Tabelle 10.13 finden Sie eine kurze Beschreibung der wichtigsten Methoden, die im anschließenden Beispiel dann zur Anwendung kommen.

Methode	Beschreibung

`bool AddPage(`
`  wxNotebookPage* page,`
`  const wxString& text,`
`  bool select = false,`
`  int imageId = -1 );`
	Fügt eine neue Seite ein. Die Seite wird mit `page` (meistens vom Typ `wxPanel`) angegeben. Der Text, den das Register erhalten soll, wird mit `text` angegeben. Wenn Sie wollen, dass diese Seite gleich ausgewählt wird, müssen Sie `true` angeben, ansonsten `false`. Mit `imageID` können Sie optional einen Image-Index für die neue Seite angeben.

Tabelle 10.13 Einige Methoden von »wxNotebook«

Methode	Beschreibung
`bool InsertPage( size_t index, wxNotebookPage* page, const wxString& text, bool select = false, int imageId = -1 );`	Wie `AddPage`, nur wird durch den zusätzlichen ersten Parameter `index` die Position (angefangen bei 0) angegeben, an der die neue Seite erzeugt werden soll.
`bool DeletePage(size_t page);`	Löscht eine Seite, die mit `page` angegeben wurde (angefangen bei 0) mit den darin enthaltenen Elementen (also auch `wxPanel`).
`size_t GetPageCount() const;`	Gibt die Anzahl der Seiten in der Registerkarte (`wxNotebook`) zurück.
`wxString GetPageText( size_t nPage ) const;`	Ermittelt den String für die Seite `page`.
`bool SetPageText( size_t page, const wxString& text );`	Setzt den Text `text` für die Seite `page`.

Tabelle 10.13 Einige Methoden von »wxNotebook« (Forts.)

Hierzu soll wieder ein Beispiel erstellt werden. Zunächst erzeugen wir in einem `wxFrame` ein `wxNotebook` mit zwei Registern. Jedes dieser Register enthält im Grunde nur Buttons (`wxButton`), die auch mit einem Ereignis-Handle verknüpft wurden. Außerdem können Sie über das Menü DATEI neue Reiter vorn oder hinten hinzufügen und auch löschen. Zunächst die Header-Datei *base.h*:

```
// base.h
#ifndef BASIC_H
#define BASIC_H

// Für die Ereignis-Handles
enum {
   MENU_FILE_QUIT,
   MENU_FILE_ADD,
   MENU_FILE_INSERT,
   MENU_FILE_DELETE,
   wxID_Button1,
   wxID_Button2,
   wxID_Button3,
};

class wxDialogDemoApp : public wxApp {
   public: virtual bool OnInit();
};
```

```
class BasicFrame : public wxFrame {
private:
   wxMenuBar *MenuBar;
   wxMenu    *ExampleMenu;
   wxNotebook* notebook;
   // Ereignis-Tabelle einrichten
   DECLARE_EVENT_TABLE()
public:
   BasicFrame( const wxChar *title,
               int xpos, int ypos,
               int width, int height);
   ~BasicFrame();

   // Methoden, die auf Ereignisse reagieren
   void OnMenuFileQuit(wxCommandEvent &event);
   void OnMenuFileAddPage(wxCommandEvent& event);
   void OnMenuFileInsertPage(wxCommandEvent& event);
   void OnMenuFileDeletePage(wxCommandEvent& event);
   void OnButton1(wxCommandEvent &event);
   void OnButton2(wxCommandEvent &event);
   void OnButton3(wxCommandEvent &event);
};

// Globale Funktionen
wxPanel *CreatePanelWithButton(wxNotebook *parent);
wxPanel *CreatePanelWithTwoButton(wxNotebook *parent);
#endif
```

Jetzt die Quelldatei *base.cpp*:

```
// base.cpp
#include <wx/wx.h>
#include <wx/notebook.h>
#include "base.h"

IMPLEMENT_APP(wxDialogDemoApp)

bool wxDialogDemoApp::OnInit() {
   BasicFrame *frame =
      new BasicFrame(
         wxT("Demonstriert wxNotebook und wxPanel"),
         50, 50, 400, 300 );
   frame->Show(TRUE);
   SetTopWindow(frame);
   return TRUE;
}
```

```
BasicFrame::BasicFrame (
   const wxChar *title,
   int xpos, int ypos,
   int width, int height)
   : wxFrame ( (wxFrame *) NULL,
               -1, title,
               wxPoint(xpos, ypos),
               wxSize(width, height),wxDEFAULT_FRAME_STYLE)
{
   // Eine Menübar erzeugen
   MenuBar = new wxMenuBar();
   // Ein Menü erzeugen
   ExampleMenu = new wxMenu();
   ExampleMenu->Append( MENU_FILE_ADD,
                        wxT("Neues Tab(hinten)"));
   ExampleMenu->Append( MENU_FILE_INSERT,
                        wxT("Neues Tab(vorne)"));
   ExampleMenu->Append( MENU_FILE_DELETE,
                        wxT("Tab loeschen") );
   ExampleMenu->Append( MENU_FILE_QUIT, wxT("&Beenden"));
   MenuBar->Append( ExampleMenu, wxT("&Datei"));
   SetMenuBar(MenuBar);

   // Ein Notebook erzeugen
   notebook = new wxNotebook(
      this, wxID_ANY, wxDefaultPosition, wxSize(300, 200),
      wxNB_MULTILINE | wxNB_FIXEDWIDTH);

   // Zwei neue Seiten der Klasse wxPanel erzeugen
   wxPanel* window1 = CreatePanelWithButton(notebook);
   wxPanel* window2 = CreatePanelWithTwoButton(notebook);
   // ... und zum Notebook hinzufügen
   notebook->AddPage(window1, wxT("Tab No. 1"), true);
   notebook->AddPage(window2, wxT("Tab No. 2"), false);
   // Eine Statusleiste
   CreateStatusBar(1);
}

BasicFrame::~BasicFrame() { }

// Ereignis-Tabelle für das Frame
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   EVT_MENU(MENU_FILE_QUIT, BasicFrame::OnMenuFileQuit)
   EVT_MENU(MENU_FILE_ADD, BasicFrame::OnMenuFileAddPage)
   EVT_MENU(
      MENU_FILE_INSERT,BasicFrame::OnMenuFileInsertPage)
```

```
   EVT_MENU(
      MENU_FILE_DELETE, BasicFrame::OnMenuFileDeletePage)
   EVT_BUTTON(wxID_Button1, BasicFrame::OnButton1)
   EVT_BUTTON(wxID_Button2, BasicFrame::OnButton2)
   EVT_BUTTON(wxID_Button3, BasicFrame::OnButton3)
END_EVENT_TABLE()

// Ereignis-Handles, die aufgerufen werden ...

// ... wenn im Menü "Beenden" ausgewählt wurde
void BasicFrame::OnMenuFileQuit(wxCommandEvent &event) {
   Destroy();
}

// ... wenn im Menü "Neues Tab (hinten)" ausgewählt wurde
void BasicFrame::OnMenuFileAddPage(wxCommandEvent &event) {
   wxPanel *panel = new wxPanel(notebook, wxID_ANY);
   // Text-Label erzeugen und zum Panel hinzufügen
   (void) new wxStaticText(
      panel, -1, wxT("Ein neues leeres Tab erzeugt"),
      wxPoint(10, 10), wxSize(200, 200), wxALIGN_CENTRE);
   // Neue Seite hinten hinzufügen
   notebook->AddPage(
      panel, wxT("Neues Tab (hinten)"), true );
   // Text in Statusleiste
   SetStatusText(wxT("Neues leeres Tab wurde erzeugt"));
}

// ... wenn im Menü "Neues Tab (vorne)" ausgewählt wurde
void BasicFrame::OnMenuFileInsertPage(wxCommandEvent &event)
{
   wxPanel *panel = new wxPanel(notebook, wxID_ANY);
   // Text-Label erzeugen und zum Panel hinzufügen
   (void) new wxStaticText(
      panel, -1, wxT("Ein neues leeres Tab erzeugt"),
      wxPoint(10, 10), wxSize(200, 200), wxALIGN_CENTRE);
   // Neue Seite vorne einfügen
   notebook->InsertPage(
      0, panel, wxT("Neues Tab (vorne)"), true);
   // Text in Statusleiste
   SetStatusText(wxT("Neues leeres Tab wurde erzeugt"));
}

// ... wenn im Menü "Tab loeschen" ausgewählt wurde
void BasicFrame::OnMenuFileDeletePage(
   wxCommandEvent& event)
```

```
{

   // Anzahl der Seiten ermitteln
   size_t n = notebook->GetPageCount();
   // Noch was zum Löschen vorhanden
   if ( n > 0 ) {
      // Eine Seite vorne löschen
      notebook->DeletePage(0);
      SetStatusText(wxT("Ein Tab geloescht"));
   }
   else {
      SetStatusText(wxT("Nix mehr zum loeschen da"));
   }
}

void BasicFrame::OnButton1(wxCommandEvent &event) {
   SetStatusText(wxT("Button1 wurde gedrückt"));
}

void BasicFrame::OnButton2(wxCommandEvent &event) {
   SetStatusText(wxT("Button2 wurde gedrückt"));
}

void BasicFrame::OnButton3(wxCommandEvent &event) {
   SetStatusText(wxT("Button3 wurde gedrückt"));
}

// Erzeugt ein Panel mit einem Button und gibt dies zurück
wxPanel *CreatePanelWithButton(wxNotebook *parent) {
    wxPanel *panel = new wxPanel(parent, wxID_ANY);
    (void) new wxButton( panel, wxID_Button1,
        wxT("Button1"), wxPoint(10, 10), wxDefaultSize );
    return panel;
}

// Erzeugt ein Panel mit zwei Buttons und gibt dies zurück
wxPanel *CreatePanelWithTwoButton(wxNotebook *parent) {
    wxPanel *panel = new wxPanel(parent, wxID_ANY);
    (void) new wxButton( panel, wxID_Button2,
        wxT("Button2"), wxPoint(10, 10), wxDefaultSize );
    (void) new wxButton( panel, wxID_Button3,
        wxT("Button3"), wxPoint(10, 40), wxDefaultSize );
    return panel;
}
```

Das Programm bei der Ausführung (siehe Abbildungen 10.15 und 10.16):

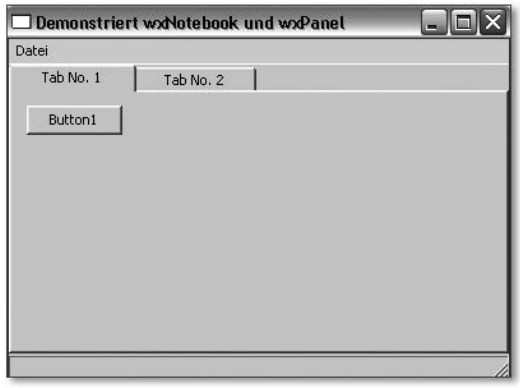

Abbildung 10.15 Beim Start des Programms (»wxNotebook« und »wxPanel«)

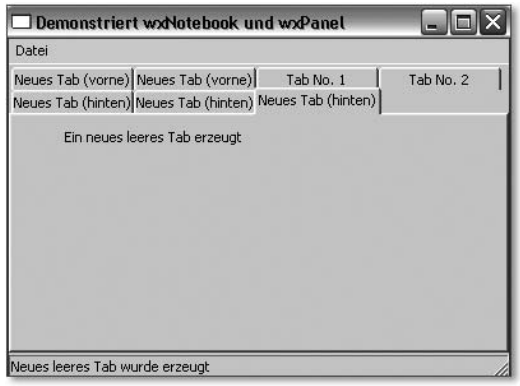

Abbildung 10.16 Nach dem Erzeugen mehrerer Register

Alternativen zu »wxNotebook«

`wxNotebook` ist abgeleitet von der Basisklasse `wxBookCtrlBase`. Von dieser Basisklasse sind neben `wxNotebook` vier weitere Varianten verfügbar, und zwar die Klassen `wxListbook`, `wxChoicebook`, `wxTreebook` und `wxToolbook`.

`wxListbook` ist `wxNotebook` recht ähnlich, aber es verwendet die Klasse `wxListCtrl` und zeigt Text-Labels statt Register (Tabs) an. Zwar gibt es für `wxListbook` noch keine richtige Dokumentation, aber die Verwendung ist der von `wxNotebook` gleichwertig. Am besten sehen Sie sich das Programmbeispiel zu `wxNotebook` an, das gewöhnlich verfügbar ist, wenn Sie `wxWidgets` installiert haben. Außerdem löst die Verwendung von `wxListCtrl` das Problem bei Mac OS, wo die Anzahl der Register beschränkt ist.

Das Beispiel von `wxNotebook` sieht umgeschrieben auf `wxListbook` folgendermaßen aus:

Abbildung 10.17 »wxListbook« im Einsatz

Hinweis [«]

Wenn Sie auch das Beispiel zu `wxNotebook` umschreiben wollen, damit `wxListbook` benutzt wird, müssen Sie nur alle Klassennamen von `wxNotebook` umändern in `wxListbook` und die Header-Datei *<wx/listbook.h>* mit einbinden.

`wxChoicebook` ist wiederum der Klasse `wxNotebook` recht ähnlich, nur wurde dies mit `wxChoice` mit einer Dropdown-Liste statt Registern (Tabs) implementiert. Auch hier ist die Anwendung im Grunde dieselbe wie schon bei `wxNotebook`. `wxChoicebook` wird gewöhnlich dort verwendet, wo wenig Platz auf dem Bildschirm vorhanden ist (beispielsweise bei Smartphones). Auf unser Beispiel bezogen sieht `wxChoicebook` folgendermaßen aus:

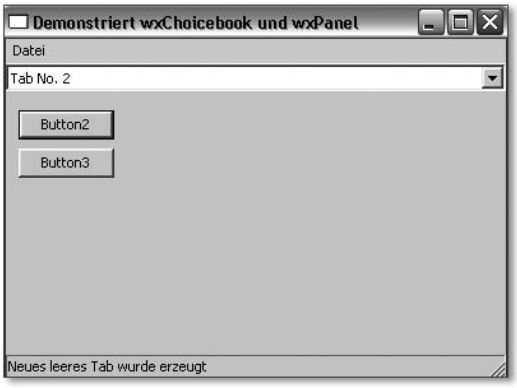

Abbildung 10.18 »wxChoicebook« im Einsatz

[»] **Hinweis**

Wollen Sie auch das Beispiel zu `wxNotebook` umschreiben, damit `wxChoicebook` benutzt wird, müssen Sie nur alle Typen von `wxNotebook` umändern in `wxChoicebook` und die Header-Datei *<wx/choicebk.h>* mit einbinden.

Bei den anderen beiden Klassen, `wxTreebook` und `wxToolbook`, wird man leider feststellen müssen, dass diese häufig gar nicht in `wxWidgets` implementiert sind. Bei `wxTreebook` handelt es sich um eine Erweiterung von `wxNotebook` in Listenform. Nur kann hierbei eine baumähnliche Struktur verwendet werden. Dasselbe gilt für `wxToolbook`, das statt der Register wie bei `wxNotebook` einfache Text-Labels verwendet.

wxScrolledWindow

Jedes Fenster kann theoretisch eine Scrollbar haben, wenn der Platz nicht mehr ausreicht. Wollen Sie aber immer eine Scrollbar anzeigen, müssen Sie `wxScrolledWindow` dazu verwenden. Damit können Sie eine Scrollbar implementieren und vorgeben, wie viel auf einmal und insgesamt gescrollt werden kann (sowohl vertikal als auch horizontal). Bei einigen Anwendungen, wie einem Text-Editor, einer Tabellenkalkulation usw., sollte man es aber dem Element selbst überlassen, wie weit und wie viel man scrollen kann, weil dies von der Menge und der Anzahl der Daten abhängig ist. Es ist also nicht immer ratsam, das Scrollen selbst zu implementieren. Hier die Syntax des Konstruktors, mit der Sie ein scrollendes Fenster erzeugen:

```
wxScrolledWindow( wxWindow* parent,
                  wxWindowID id = -1,
                  const wxPoint& pos = wxDefaultPosition,
                  const wxSize& size = wxDefaultSize,
                  long style = wxHSCROLL | wxVSCROLL,
                  const wxString& name = "scrolledWindow" );
```

Die einzelnen Parameter von `wxScrolledWindow` haben folgende Bedeutung:

- `parent` – das Eltern-Fenster (darf nicht `NULL` sein)
- `id` – die Identifizierung des Fensters
- `pos` – die Position; bei (–1, –1) wird wieder die Default-Position verwendet.
- `size` – die Größe; bei (–1, –1) wird wieder die Default-Größe verwendet.
- `style` – der Fensterstil; eigentlich gibt es keine speziellen Styles, aber gewöhnlich können Sie hierfür `wxVSCROLL | wxHSCROLL` hinzufügen.
- `name` – der Name für das Panel (wird nur unter Motif benötigt)

Nach dem Erzeugen eines Fensters wird allerdings noch keine sichtbare Scrollbar darin erstellt. Wenn Sie diese anzeigen wollen, müssen Sie die Methode `wxScrolledWindow::SetScrollbar` verwenden. Hierzu ein kurzer Überblick über zwei recht nützliche Methoden von `wxScrolledWindow`:

Methode	Beschreibung
`void EnableScrolling( const bool xScrolling, const bool yScrolling );`	Damit können Sie das Scrolling in eine bestimmte Richtung ein- bzw. abschalten. Mit `true` schalten Sie das Scrollen in der entsprechenden Richtung ein und mit `false` ab.
`void SetScrollbars( int pixelsPerUnitX, int pixelsPerUnitY, int noUnitsX, int noUnitsY, int xPos = 0, int yPos = 0, bool noRefresh = false );`	Damit stellen Sie die vertikale und/oder horizontale Scrollbar ein. Mit `pixelsPerUnitX` bzw. `pixelsPerUnitY` geben Sie die Anzahl der Pixel an, die bei einmal Scrollen in die horizontale bzw. vertikale Richtung gescrollt werden. »Einmal Scrollen« ist beispielsweise das Drücken der Pfeiltaste nach unten oder das Klicken mit der Maus auf den Pfeil der Scrollbar (oder auch das Mausrad). Mit `noUnitsX` und `noUnitsY` geben Sie die Anzahl (oder auch Größe) der Einheit an, die in die vertikale bzw. horizontale Richtung gescrollt werden kann. Mit `xPos` und `yPos` können Sie die Position der Scrollbar (genauer des Scrollbalkens) in der vertikalen und horizontalen Richtung angeben, an der diese positioniert werden soll. Wird `refresh` nicht auf `true` gesetzt, wird das Fenster nicht erneuert.

Tabelle 10.14 Einige Methoden von »wxScrolledWindow«

`wxScrolledWindow` generiert auch ein Ereignis (`wxScrollWinEvent`). Leider besteht hierbei keine Möglichkeit, eine Eltern-Kind-Beziehung (beispielsweise zwischen `wxFrame` und `wxScrolledWindow`) herzustellen, so dass man gezwungen ist – wenn man die Ereignisse behandeln will –, eine neue Klasse zu erzeugen und `wxScrolledWindow` als Basisklasse zu verwenden (ähnlich wie Sie dies bereits mit `wxFrame` gemacht haben). Die folgende Tabelle (10.15) listet auf, was Sie alles behandeln können und wie Sie hierfür einen Handle in die Ereignis-Tabelle eintragen.

Handle einrichten	Beschreibung
`EVT_SCROLLWIN(func)`	Behandelt alle Scroll-Ereignisse.
`EVT_SCROLLWIN_TOP(func)`	Behandelt `wxEVT_SCROLLWIN_TOP`-Ereignisse (Scrollbalken ganz oben = Minimum-Position).
`EVT_SCROLLWIN_BOTTOM(func)`	Behandelt `wxEVT_SCROLLWIN_BOTTOM`-Ereignisse (Scrollbalken ganz unten = Maximum-Position).

Tabelle 10.15 »wxScrolledWindow«-Ereignisse behandeln

Handle einrichten	Beschreibung
EVT_SCROLLWIN_LINEUP(func)	Behandelt wxEVT_SCROLLWIN_LINEUP-Ereignisse (eine »Zeile« nach oben).
EVT_SCROLLWIN_LINEDOWN(func)	Behandelt wxEVT_SCROLLWIN_LINEDOWN-Ereignisse (eine »Zeile« nach unten).
EVT_SCROLLWIN_PAGEUP(func)	Behandelt wxEVT_SCROLLWIN_PAGEUP-Ereignisse (eine »Seite« nach oben).
EVT_SCROLLWIN_PAGEDOWN(func)	Behandelt wxEVT_SCROLLWIN_PAGEDOWN-Ereignisse (eine »Seite« nach unten).

Tabelle 10.15 »wxScrolledWindow«-Ereignisse behandeln (Forts.)

Zur Demonstration soll nochmals das Beispiel mit wxNotebook verwendet werden. Hierbei sollen jetzt drei Register mit wxScrolledWindow erzeugt werden. Dazu wurde ein Register mit einer horizontalen, eines mit einer vertikalen sowie eines mit einer horizontalen und vertikalen Scrollbar erzeugt. Damit Sie auch gleich sehen können, wie man auf Ereignisse von wxScrolledWindow reagieren kann, wurde eine eigene Klasse MyScrolledWindow erstellt. In dem Beispiel wird ein Ereignis-Handle eingerichtet, der auf alle Scroll-Ereignisse reagiert. Zunächst wieder die Header-Datei *base.h*:

```
// base.h
#ifndef BASIC_H
#define BASIC_H

// Für die Ereignis-Handles
enum {
   MENU_FILE_QUIT,
   ID_FRAME
};

class wxDialogDemoApp : public wxApp {
   public: virtual bool OnInit();
};

class BasicFrame : public wxFrame {
private:
   wxMenuBar *MenuBar;
   wxMenu    *ExampleMenu;
   wxNotebook* notebook;
   // Ereignis-Tabelle einrichten
   DECLARE_EVENT_TABLE()
public:
   BasicFrame( const wxChar *title,
```

```
                int xpos, int ypos,
                int width, int height);
   ~BasicFrame();

   // Methoden, die auf Ereignisse reagieren
   void OnMenuFileQuit(wxCommandEvent &event);
   void OnScrolledWindow( wxScrollWinEvent& event );
};

class MyScrolledWindow : public wxScrolledWindow {
private:
   DECLARE_EVENT_TABLE()
public:
   MyScrolledWindow( wxWindow* parent, wxWindowID id,
                     int xpos, int ypos,
                     int width, int height,
                     long style, int pixelPerUnitX,
                     int pixelPerUNIXY, int noUnitsX,
                     int noUnitsY, bool refresh );
   ~MyScrolledWindow();
   void OnScrolledWindow(wxScrollWinEvent& event );
};
#endif
```

Jetzt die Quelldatei *base.cpp*:

```
// base.cpp
#include <wx/wx.h>
#include <wx/notebook.h>
#include "base.h"

IMPLEMENT_APP(wxDialogDemoApp)

bool wxDialogDemoApp::OnInit() {
   BasicFrame *frame =
      new BasicFrame( wxT("Demonstriert wxScrollbar"),
                      50, 50, 400, 300 );
   frame->Show(TRUE);
   SetTopWindow(frame);
   return TRUE;
}

BasicFrame::BasicFrame (
   const wxChar *title,
   int xpos, int ypos,
```

```
    int width, int height)
    : wxFrame ( (wxFrame *) NULL,
                ID_FRAME, title,
                wxPoint(xpos, ypos),
                wxSize(width, height),wxDEFAULT_FRAME_STYLE)
{
    // Eine Menübar erzeugen
    MenuBar = new wxMenuBar();
    // Ein Menü erzeugen
    ExampleMenu = new wxMenu();
    ExampleMenu->Append(MENU_FILE_QUIT, wxT("&Beenden"));
    MenuBar->Append( ExampleMenu, wxT("&Datei"));
    SetMenuBar(MenuBar);

    // Ein Notebook erzeugen
    notebook = new wxNotebook(
       this, wxID_ANY, wxDefaultPosition, wxSize(300, 200));
    // Ein Fenster zum Scrollen nur in vertikaler Richtung
    MyScrolledWindow *window1 = new MyScrolledWindow(
       notebook, wxID_ANY, 0, 0, -1, -1,
       wxVSCROLL, 0, 10, 0, 500, true );
    // Einen Button hinzufügen
    (void) new wxButton( window1, wxID_ANY, wxT("Button1"),
       wxPoint(10, 10), wxDefaultSize );
    // Ein Fenster zum Scrollen nur in horizontaler Richtung
    MyScrolledWindow *window2 = new MyScrolledWindow(
       notebook, wxID_ANY, 0, 0, -1, -1,
       wxHSCROLL, 10, 0, 500, 0, true );
    // Einen Button hinzufügen
    (void) new wxButton(window2 , wxID_ANY, wxT("Button2"),
        wxPoint(10, 10), wxDefaultSize );
    // Ein Fenster zum Scrollen in beide Richtungen
    MyScrolledWindow *window3 = new MyScrolledWindow(
       notebook, wxID_ANY, 0, 0, -1, -1,
       wxHSCROLL|wxVSCROLL, 10, 10, 500, 500, true );
    // Einen Button hinzufügen
    (void) new wxButton(window3,  wxID_ANY, wxT("Button3"),
        wxPoint(10, 10), wxDefaultSize );

    // ... und zum Notebook hinzufügen
    notebook->AddPage(
       window1, wxT("Vertikal Scroll"), true );
    notebook->AddPage(
       window2, wxT("Horizontal Scroll"), false);
    notebook->AddPage(
```

```
      window3, wxT("Vertikal und Horizontal"), false);
   // Eine Statusleiste

   CreateStatusBar(1);
}

BasicFrame::~BasicFrame() { }

// Ereignis-Tabelle für das Frame
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   EVT_MENU(MENU_FILE_QUIT, BasicFrame::OnMenuFileQuit)
END_EVENT_TABLE()

void BasicFrame::OnMenuFileQuit(wxCommandEvent &event) {
   Destroy();
}

MyScrolledWindow::MyScrolledWindow(
                     wxWindow* parent, wxWindowID id,
                     int xpos, int ypos,
                     int width, int height,
                     long style, int pixelPerUnitX,
                     int pixelPerUnitY, int noUnitsX,
                     int noUnitsY, bool refresh ):
   wxScrolledWindow( parent, id, wxPoint(xpos, ypos),
                      wxSize(width, height), style )
{
   SetScrollbars( pixelPerUnitX, pixelPerUnitY,
                  noUnitsX, noUnitsY,
                  0, 0, refresh );
}

MyScrolledWindow::~MyScrolledWindow() {}

void MyScrolledWindow::OnScrolledWindow(
   wxScrollWinEvent& event ) {
   // Um die Statuszeile nutzen zu können, benötigen wir
   // einen Weg, um darauf zuzugreifen
   // (siehe wxWindow::FindWindowbyID)
   wxFrame* win = (wxFrame*) FindWindowById(ID_FRAME, NULL);
   // wurde vertikal gescrollt
   // siehe auch wxScrollWinEvent::GetOrientation
   if( event.GetOrientation() == wxVERTICAL )
      win->SetStatusText(
         wxT("Scrollbar vertikal betätigt") );
```

```
   // oder wars horizontal
   else if( event.GetOrientation() == wxHORIZONTAL )
      win->SetStatusText(
         wxT("Scrollbar horizontal betätigt") );
   // ... weitermachen ohne besondere Änderungen
   event.Skip();
}

// Ereignis-Handle für wxScrolledWindow
BEGIN_EVENT_TABLE(MyScrolledWindow, wxScrolledWindow)
   EVT_SCROLLWIN(MyScrolledWindow::OnScrolledWindow)
END_EVENT_TABLE()
```

Das Programm bei der Ausführung:

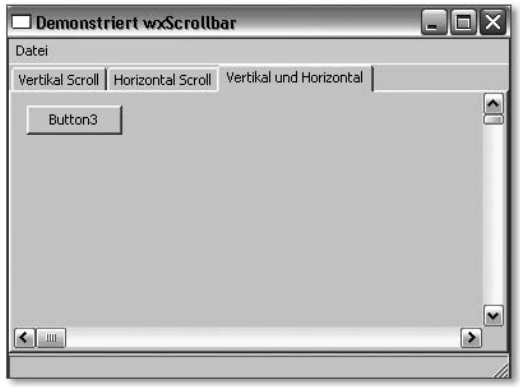

Abbildung 10.19 »wxScrolledWindow« im Einsatz

[»] **Hinweis**

Wenn Sie ein eigenes Scrolling etwas ungewöhnlicher implementieren wollen, können Sie auch eine Klasse erstellen, diese von `wxWindow` ableiten und die darin enthaltene Klasse `wxWindow::SetScrollbar` verwenden.

[»] **Hinweis**

Eine weitere Klasse zum Implementieren des Scrollings ist `wxVScrolledWindow`.

wxSplitterWindow

Die Klasse `wxSplitterWindow` verwaltet zwei aufgeteilte Unterfenster. Wenn Sie diese Klasse »verschachteln«, können selbstverständlich auch mehrere Unterfenster verwendet werden. Der Konstruktor von `wxSplitterWindow` hat folgende Syntax:

```
wxSplitterWindow( wxWindow* parent,
                  wxWindowID id,
                  const wxPoint& point = wxDefaultPosition,
                  const wxSize& size = wxDefaultSize,
                  long style=wxSP_3D,
                  const wxString& name = "splitterWindow" );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster von `wxSplitterWindow`
- `id` – die Identifizierung des Fensters
- `pos` – die Position; bei (–1, –1) wird wieder die Default-Position verwendet.
- `size` – die Größe; bei (–1, –1) wird wieder die Default-Größe verwendet.
- `style` – der Fensterstil (siehe Tabelle 10.16)
- `name` – der Name für das Fenster (wird nur unter Motif benötigt)

Stil	Beschreibung
`wxSP_3D`	Zeichnet einen 3D-Effekt am Fensterrahmen und Schieberahmen (Trennleiste).
`wxSP_3DSASH`	Zeichnet einen 3D-Effekt am Schieberahmen (Trennleiste).
`wxSP_3DBORDER`	Zeichnet einen 3D-Effekt am Fensterrahmen.
`wxSP_BORDER`	Zeichnet einen Standardfensterrahmen.
`wxSP_NOBORDER`	kein spezieller Rahmen (Standardeinstellung)
`wxSP_NO_XP_THEME`	Fügt einen versunkenen Fensterrahmen und einen 3D-Effekt am Schieberahmen ein.
`wxSP_LIVE_UPDATE`	Verändert die Größe des Kind-Fensters sofort, wenn die Größe des Schieberahmens verändert wurde.

Tabelle 10.16 Verschiedene Fensterstile von »wxSplitterWindow«

Wenn Sie diesen Konstruktor aufgerufen haben, müssen Sie entweder ein oder zwei Unterfenster (beispielsweise `wxScrolledWindow` oder `wxPanel`) erzeugen, die das erzeugte `wxSplitterWindow`-Fenster als Eltern-Fenster erhalten. Anschließend muss eine der Methoden `wxSplitterWindow::Initialize`, `wxSplitterWindow::SplitVertically` oder `wxSplitterWindow::SplitHorizontally` aufgerufen werden, um die einzelnen Fenster zu ordnen.

Natürlich können Sie hierbei zwei Fenster erzeugen, von denen eins versteckt ist und nicht gleich angezeigt wird, oder Sie erzeugen ein zweites Fenster nachträglich auf Anfrage (dynamisch).

`wxSplitterWindow` generiert ein `wxSplitterEvent`-verbreitendes Ereignis. Wie Sie einen Handle dafür einrichten, können Sie aus Tabelle 10.17 ablesen.

Handle einrichten	Beschreibung
`EVT_SPLITTER_SASH_POS_CHANGING( id, func )`	Wird von einem `wxEVT_COMMAND_SPLITTER_SASH_ POS_CHANGING`-Ereignis ausgelöst und generiert, wenn die Position des Schieberahmens gerade verändert wird. Hierbei können Sie zum Beispiel `Veto` aufrufen, um die Veränderung der Position zu stoppen.
`EVT_SPLITTER_SASH_POS_CHANGED( id, func )`	Wird von einem `wxEVT_COMMAND_SPLITTER_SASH_POS_CHANGED`-Ereignis ausgelöst und generiert, wenn sich die Position des Schieberahmens bereits verändert hat.
`EVT_SPLITTER_UNSPLIT(id, func)`	Wird von einem `wxEVT_COMMAND_SPLITTER_UNSPLIT`-Ereignis ausgelöst und generiert, wenn die Aufteilung aufgehoben wird (`Unsplit`).
`EVT_SPLITTER_DCLICK(id, func)`	Wird von einem `wxEVT_COMMAND_SPLITTER_DOUBLECLICKED`-Ereignis ausgelöst und generiert, wenn der Schieberahmen doppelt angeklickt wurde.

Tabelle 10.17 Ereignisse von »wxSplitterWindow«

Die Klasse `wxSplitterWindow` enthält eine Reihe sehr interessanter Methoden, von denen die wichtigsten hier beschrieben werden:

Methode	Beschreibung
`bool IsSplit() const;`	Gibt `true` zurück, wenn das Fenster bereits aufgeteilt ist, ansonsten wird `false` zurückgegeben.
`void SetSashPosition( int position, const bool redraw = true );`	Setzt die Position des Schieberahmens auf die Position `position` (in Pixeln). Wenn `redraw` auf `true` gesetzt ist, wird bei Veränderung der Größe, des Fenster- und Schieberahmens neu gezeichnet. Achtung, diese Methode überprüft leider nicht einen Wert außerhalb des Bereichs.
`void SetMinimumPaneSize( int paneSize);`	Legt mit `paneSize` die Mindestgrenze für die beiden Fenster fest. Der Standardwert ist hierbei `0`, was bedeutet, dass ein Fenster bis auf `0` reduziert werden kann. Das bedeutet auch, dass hiermit das Fenster komplett entfernt wurde. Um dem entgegenzuwirken, können Sie eine Mindestgrenze angeben, oder Sie reagieren auf ein `wxEVT_COMMAND_SPLITTER_UNSPLIT`-Ereignis und legen wieder ein `Veto` ein.

Tabelle 10.18 Einige Methoden von »wxSplitterWindow«

Methode	Beschreibung
`bool SplitHorizontally ( wxWindow* window1, wxWindow* window2, int sashPosition = 0 );`	Initialisiert den oberen und unteren Teil des aufgeteilten Fensters. Mit `window1` geben Sie das obere und mit `window2` das untere Fenster an. Die Position der Schiebeleiste geben Sie mit `sashPosition` an. Ist der Wert positiv, wird die Größe vom oberen Fenster aus bezogen. Bei einem negativen Wert erfolgt die Größe vom unteren Fenster. Verwenden Sie `0` (Standardwert), haben beide Fenster die gleiche Größe (Hälfte – Hälfte). Bevor Sie diese Methode aufrufen, sollten Sie mit `IsSplit` überprüfen, ob das Fenster bereits geteilt wurde.
`bool SplitVertically( wxWindow* window1, wxWindow* window2, int sashPosition = 0 );`	Initialisiert den linken und rechten Teil des aufgeteilten Fensters. Mit `window1` geben Sie das linke und mit `window2` das rechte Fenster an. Die Position der Schiebeleiste geben Sie mit `sashPosition` an. Ist der Wert positiv, wird die Größe vom linken Fenster aus bezogen. Bei einem negativen Wert erfolgt die Größe vom rechten Fenster. Verwenden Sie `0` (Standardwert), haben beide Fenster die gleiche Größe (Hälfte – Hälfte). Bevor Sie diese Methode aufrufen, sollten Sie mit `IsSplit` überprüfen, ob das Fenster bereits geteilt wurde.
`bool Unsplit( wxWindow* toRemove = NULL );`	Damit können Sie die Aufteilung des Fensters aufheben. Das Fenster, das Sie entfernen wollen, wird mit `toRemove` angegeben. Das Fenster wird allerdings nicht wirklich gelöscht, sondern nur im Hintergrund gehalten.
`wxWindow* GetWindow1() const;`	Gibt einen Zeiger auf das linke oder obere Fenster zurück.
`wxWindow* GetWindow2() const;`	Gibt einen Zeiger auf das rechte oder untere Fenster zurück.

Tabelle 10.18 Einige Methoden von »wxSplitterWindow« (Forts.)

Es folgt nun ein Beispiel, das die hier beschriebene Klasse `wxSplitterWindow` in der Praxis zeigen soll. Dabei ist es möglich, das Fenster horizontal, vertikal oder gar nicht aufzuteilen. Zudem kann die Mindestgröße für ein Fenster festgelegt werden.

Hinweis **[«]**

In dem Beispiel wird mit `EVT_UDPATE_UI` (aus der Klasse `wxUpdateUIEvent`) auch ein spezieller Ereignis-Handle eingerichtet, der im Allgemeinen dazu eingesetzt wird, eine Benutzerschnittstelle wie Menüs, Toolbars usw. zu erneuern. In dem Beispiel wird die Methode `Enable` verwendet, um einzelne Benutzerschnittstellen einzuschalten. Genaueres können Sie der Dokumentation zu `wxUpdateEvent` entnehmen.

Es wurde zudem eine Methode eingesetzt, die einen Device-Context verwendet und damit etwas auf dem Bildschirm zeichnet (in diesem Fall etwas auf dem Bildschirm schreibt).

Die Header-Datei *base.h*:

```
// base.h
#ifndef BASIC_H
#define BASIC_H

// Identifizierer für das Menü
enum {
    SPLIT_QUIT = 1,
    SPLIT_HORIZONTAL,
    SPLIT_VERTICAL,
    SPLIT_UNSPLIT,
    SPLIT_SETMINSIZE
};

class MyApp: public wxApp {
public:
   MyApp() { }
   virtual bool OnInit();
};


class MyFrame: public wxFrame {
private:
   wxScrolledWindow *m_left, *m_right;
   wxSplitterWindow* m_splitter;
   wxWindow *m_replacewindow;
   DECLARE_EVENT_TABLE()
public:
   MyFrame();
   virtual ~MyFrame();

   // Menü-Kommandos
   void SplitHorizontal(wxCommandEvent& event);
   void SplitVertical(wxCommandEvent& event);
   void Unsplit(wxCommandEvent& event);
   void SetMinSize(wxCommandEvent& event);
   void Quit(wxCommandEvent& event);

   // Update-Funktionen
   void UpdateUIHorizontal(wxUpdateUIEvent& event);
```

```
   void UpdateUIVertical(wxUpdateUIEvent& event);
   void UpdateUIUnsplit(wxUpdateUIEvent& event);
};

class MySplitterWindow : public wxSplitterWindow {
private:
   wxFrame *m_frame;
   DECLARE_EVENT_TABLE()
public:
   MySplitterWindow(wxFrame *parent);
   // Ergeignis-Handle
   void OnPositionChanged(wxSplitterEvent& event);
   void OnPositionChanging(wxSplitterEvent& event);
   void OnUnsplitEvent(wxSplitterEvent& event);
};

class MyCanvas: public wxScrolledWindow {
private:
   bool m_mirror;
public:
   MyCanvas(wxWindow* parent, bool mirror);
   virtual ~MyCanvas(){};
   virtual void OnDraw(wxDC& dc);
};
#endif
```

Die Quelldatei *base.cpp*:

```
// base.cpp
#include <wx/wx.h>
#include <wx/splitter.h>
#include <wx/dcmirror.h>
#include "base.h"

IMPLEMENT_APP(MyApp)

bool MyApp::OnInit() {
    MyFrame* frame = new MyFrame;
    frame->Show(true);
    return true;
}

// -----------
// MyFrame   |
// -----------
```

```
// Ereignis-Handle
BEGIN_EVENT_TABLE(MyFrame, wxFrame)
   EVT_MENU(SPLIT_VERTICAL, MyFrame::SplitVertical)
   EVT_MENU(SPLIT_HORIZONTAL, MyFrame::SplitHorizontal)
   EVT_MENU(SPLIT_UNSPLIT, MyFrame::Unsplit)
   EVT_MENU(SPLIT_SETMINSIZE, MyFrame::SetMinSize)
   EVT_MENU(SPLIT_QUIT, MyFrame::Quit)
   EVT_UPDATE_UI(SPLIT_VERTICAL, MyFrame::UpdateUIVertical)
   EVT_UPDATE_UI(
      SPLIT_HORIZONTAL, MyFrame::UpdateUIHorizontal )
   EVT_UPDATE_UI(SPLIT_UNSPLIT, MyFrame::UpdateUIUnsplit)
END_EVENT_TABLE()

// Konstruktor
MyFrame::MyFrame()
   : wxFrame( NULL, wxID_ANY,
              wxT("Demonstriert wxSplitterWindow"),
              wxDefaultPosition, wxSize(400, 300),
              wxDEFAULT_FRAME_STYLE |
              wxNO_FULL_REPAINT_ON_RESIZE)
{
   // Zwei Statusleisten
   CreateStatusBar(2);

   // Eine Menübar erzeugen
   wxMenu *splitMenu = new wxMenu;
   splitMenu->Append( SPLIT_VERTICAL,
                      wxT("Vertikal Splitten"));
   splitMenu->Append( SPLIT_HORIZONTAL,
                      wxT("Horizontal Splitten"));
   splitMenu->Append( SPLIT_UNSPLIT,
                      wxT("Splitten aufheben"));
   splitMenu->Append(SPLIT_SETMINSIZE,
                     wxT("Minimalaufteilung Einstellen"));
   splitMenu->Append(SPLIT_QUIT, wxT("Beenden"));

   wxMenuBar *menuBar = new wxMenuBar;
   menuBar->Append(splitMenu, wxT("&Splitter"));

   SetMenuBar(menuBar);
   // Ein neues wxSplitterWindow erzeugen
   m_splitter = new MySplitterWindow(this);

   // Auf der linken Seite ein wxScrolledWindow erzeugen
   m_left = new MyCanvas(m_splitter, true);
```

```
   m_left->SetBackgroundColour(*wxRED);
   m_left->SetScrollbars(20, 20, 5, 5);
   // Einen Button hinzufügen
  (void) new wxButton( m_left, wxID_ANY, wxT("Button1"),
     wxPoint(10, 10), wxDefaultSize );

   // Auf der rechten Seite ein wxScrolledWindow erzeugen
   m_right = new MyCanvas(m_splitter, false);
   m_right->SetBackgroundColour(*wxGREEN);
   m_right->SetScrollbars(20, 20, 5, 5);
   // Ebenfalls einen Button hinzufügen
  (void) new wxButton( m_right, wxID_ANY, wxT("Button2"),
     wxPoint(10, 10), wxDefaultSize );

   // Das Frame zunächst vertikal aufteilen
   m_splitter->SplitVertically(m_left, m_right, 100);
   SetStatusText(wxT("Minimale Aufteilung = 0"), 1);
}

MyFrame::~MyFrame() { }

// Ereignis-Handle für das Menü

void MyFrame::Quit(wxCommandEvent& event) {
   Close(true);
}

// Frame horizontal aufteilen
void MyFrame::SplitHorizontal(wxCommandEvent& event) {
   if ( m_splitter->IsSplit() )
      m_splitter->Unsplit();
   m_left->Show(true);
   m_right->Show(true);
   m_splitter->SplitHorizontally( m_left, m_right );
   SetStatusText(wxT(""), 0);
   SetStatusText(wxT("Aufteilung ist horizontal"), 1);
}

// Frame vertikal aufteilen
void MyFrame::SplitVertical(wxCommandEvent& event) {
   if ( m_splitter->IsSplit() )
      m_splitter->Unsplit();
   m_left->Show(true);
   m_right->Show(true);
   m_splitter->SplitVertically( m_left, m_right );
```

```
   SetStatusText(wxT(""), 0);
   SetStatusText(wxT("Aufteilung ist vertikal"), 1);
}

// Aufteilung des Frames komplett aufheben
void MyFrame::Unsplit(wxCommandEvent& event ) {
   if ( m_splitter->IsSplit() )
      m_splitter->Unsplit();
   SetStatusText(wxT("Keine Aufteilung mehr"));
}

// Minimale Grenze der Aufteilung vorgeben
void MyFrame::SetMinSize(wxCommandEvent& event) {
   wxString str;
   str.Printf(wxT("%d"), m_splitter->GetMinimumPaneSize());
   str = wxGetTextFromUser(
     wxT("Bitte Mindestgroesse eingeben:"),
     wxT(""), str, this);
   if ( str.empty() )
      return;
   int minsize = wxStrtol( str, (wxChar**)NULL, 10 );
   m_splitter->SetMinimumPaneSize(minsize);
   str.Printf(
      wxT("Mindestgroesse eines Fensters : %d"), minsize);
   SetStatusText(str, 1);
}

// Die UI-Handle erneuern
void MyFrame::UpdateUIHorizontal(wxUpdateUIEvent& event) {
   event.Enable( (!m_splitter->IsSplit()) ||
      (m_splitter->GetSplitMode() != wxSPLIT_HORIZONTAL) );
}

void MyFrame::UpdateUIVertical(wxUpdateUIEvent& event) {
   event.Enable( ( (!m_splitter->IsSplit()) ||
      (m_splitter->GetSplitMode() != wxSPLIT_VERTICAL) ) );
}

void MyFrame::UpdateUIUnsplit(wxUpdateUIEvent& event) {
   event.Enable( m_splitter->IsSplit() );
}

// --------------------
// MySplitterWindow   |
// --------------------
```

```
BEGIN_EVENT_TABLE(MySplitterWindow, wxSplitterWindow)
   EVT_SPLITTER_SASH_POS_CHANGED(
      wxID_ANY, MySplitterWindow::OnPositionChanged)
   EVT_SPLITTER_SASH_POS_CHANGING(
      wxID_ANY, MySplitterWindow::OnPositionChanging)
   EVT_SPLITTER_UNSPLIT(
      wxID_ANY, MySplitterWindow::OnUnsplitEvent)
END_EVENT_TABLE()

MySplitterWindow::MySplitterWindow(wxFrame *parent)
   : wxSplitterWindow(parent, wxID_ANY,
                      wxDefaultPosition, wxDefaultSize,
                      wxSP_PERMIT_UNSPLIT|wxSP_LIVE_UPDATE|
                      wxCLIP_CHILDREN  )
{
   m_frame = parent;
}

void MySplitterWindow::OnPositionChanged(
   wxSplitterEvent& event)
{
   wxLogStatus( m_frame,
                wxT("Pos. veraendert : %d (oder %d)"),
                event.GetSashPosition(), GetSashPosition());
   event.Skip();
}

void MySplitterWindow::OnPositionChanging(
   wxSplitterEvent& event)
{
   wxLogStatus( m_frame,
                wxT("Pos. veraendert : %d (oder %d)"),
                event.GetSashPosition(), GetSashPosition());
   event.Skip();
}

void MySplitterWindow::OnUnsplitEvent(
   wxSplitterEvent& event)
{
    m_frame->SetStatusText(wxT("Keine Aufteilung mehr"), 1),
   event.Skip();
}
```

```
// -------------
// MyCanvas     |
// -------------

MyCanvas::MyCanvas(wxWindow* parent, bool mirror)
   : wxScrolledWindow( parent, wxID_ANY,
                       wxDefaultPosition, wxDefaultSize,
                       wxHSCROLL | wxVSCROLL |
                       wxNO_FULL_REPAINT_ON_RESIZE)
{
   m_mirror = mirror;
}

// Mit wxMirrorDC einfach auf beiden Fenstern etwas zeichnen
void MyCanvas::OnDraw(wxDC& dcOrig) {
   wxMirrorDC dc(dcOrig, m_mirror);
       dc.SetFont( wxFont(12, wxMODERN,
                          wxNORMAL, wxNORMAL, true) );
   dc.DrawText( wxT("Hallo Welt"), 110, 10 );
   dc.SetBackgroundMode(wxTRANSPARENT);
   dc.SetFont( wxFont( 8, wxSWISS, wxNORMAL, wxNORMAL ) );
   wxString text;
   for ( int n = -180; n < 180; n += 30 ) {
       text.Printf(wxT("     %d Hallo Welt"), n);
       dc.DrawRotatedText(text , 110, 120, n);
   }
}
```

Das Programm bei der Ausführung (siehe Abbildungen 10.20 und 10.21):

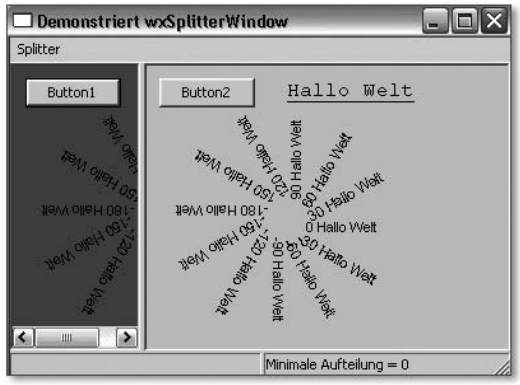

Abbildung 10.20 »wxSplitterWindow« im Einsatz (vertikal)

Abbildung 10.21 »wxSplitterWindow« im Einsatz (horizontal)

Hinweis [«]

Sofern Sie Ihr Fenster in mehrere Teile »aufsplitten« wollen/müssen, können Sie sich auch mal die Klasse `wxSashWindow` ansehen.

10.3.10 Nicht statische Kontrollelemente

Nicht statische Kontrollelemente sind Elemente wie Buttons oder Listboxen, die eine Eingabe von der Maus oder Tastatur erwarten. In diesem Abschnitt sollen die grundlegenden Kontrollelemente behandelt werden.

Hinweis [«]

Aus Platzgründen habe ich die Erzeugung der Widgets auf globale und sehr einfach gehaltene Funktionen aufgeteilt. Die einzelnen Kontrollelemente sollen dabei über ein `wxChoicebook` in einem Programm aufgerufen werden können. Auch bei den Methoden der Widgets habe ich mich auf das Nötigste beschränkt. Zumeist werden nur die Ereignisse eines Widgets behandelt, und in der Statusbox wird eine entsprechende Meldung ausgegeben. Für weitere Details kann ich Sie leider nur auf die Referenz (siehe Buch-CD) verweisen. Mit diesem Beispiel gebe ich Ihnen ein grundlegendes Konstrukt mit auf den Weg, mit dem Sie durch das Experimentieren mit den Methoden und Optionen tiefer in die Materie einsteigen können. Die Aufgabe dieses Abschnitts besteht hauptsächlich darin, die einzelnen Kontrollelemente vorzustellen.

Mit der folgenden Header-Datei *base.h* können Sie sich einen Überblick darüber verschaffen, was im Folgenden auf Sie zukommt. Sie finden das komplette Listing der Quelldatei (*base.cpp*) am Ende des Listings (und auf der Buch-CD).

```
// base.h
#ifndef BASIC_H
#define BASIC_H
```

```
#include <wx/wx.h>
#include <wx/stockitem.h>
#include <wx/bmpbuttn.h>
#include <wx/spinctrl.h>
#include <wx/tglbtn.h>
#include <wx/statline.h>
#include <wx/choicebk.h>

// Für die Ereignis-Handles
enum {
   MENU_FILE_QUIT,
   ID_BUTTON1,
   ID_CHOICE,
   ID_CHOICE_SORTED,
   ID_CHOICE_SORTED2,
   ID_CHOICE_ALL,
   ID_COMBO,
   ID_COMBO_SORTED,
   ID_COMBO_SORTED2,
   ID_COMBO_ALL,
   ID_COMBO_ADD,
   ID_CHECK_BUTTON1,
   ID_CHECK_BUTTON2,
   ID_CHECK_BUTTON3,
   ID_CHECK_BUTTON4,
   ID_LISTBOX,
   ID_CHECK_LISTBOX,
   ID_RADIOBOX,
   ID_RADIOBUTTON1,
   ID_RADIOBUTTON2,
   ID_RADIOBUTTON3,
   ID_SPINCTRL,
   ID_SLIDER,
   ID_TEXTCTRL,
   ID_TOGGLE1,
   ID_TOGGLE2,
   ID_TOGGLE3
};

class wxDialogDemoApp : public wxApp {
   public: virtual bool OnInit();
};

class BasicFrame : public wxFrame {
private:
```

```
    wxMenuBar *MenuBar;
    wxMenu    *ExampleMenu;
    wxChoicebook* choicebook;
    // Ereignis-Tabelle einrichten
    DECLARE_EVENT_TABLE()
public:
    BasicFrame( const wxChar *title,
                int xpos, int ypos,
                int width, int height);
    ~BasicFrame() {};

    // Methoden, die auf Ereignisse reagieren
    void OnMenuFileQuit(wxCommandEvent &event);
    void OnButton1(wxCommandEvent &event);
    void OnButtonOk(wxCommandEvent &event);
    void OnButtonCancel(wxCommandEvent &event);
    void OnChoice1( wxCommandEvent& event );
    void OnChoice2( wxCommandEvent& event );
    void OnCombo1( wxCommandEvent& event );
    void OnCombo2( wxCommandEvent& event );
    void OnCombo3( wxCommandEvent& event );
    void ComboText( wxCommandEvent &event );
    void OnCheck( wxCommandEvent& event );
    void OnListBox( wxCommandEvent& event );
    void OnCheckListBox( wxCommandEvent& event );
    void OnRadioBox( wxCommandEvent& event );
    void OnRadioButton( wxCommandEvent& event );
    void OnSpinCtrl( wxSpinEvent& event );
    void OnSlider( wxCommandEvent& event );
    void OnText( wxCommandEvent& event );
    void OnToggle( wxCommandEvent& event );
};

// Globale Funktionen, welche ein Panel mit einem Widget
// erzeugen und dies zurückgeben. Diese Panel packen
// wir anschließend in BasicFrame in ein wxChoicebook
wxPanel *CreatePanelWithButton(wxChoicebook *parent);
wxPanel *CreatePanelWithBitmapButton(wxChoicebook *parent);
wxPanel *CreatePanelWithChoice(wxChoicebook *parent);
wxPanel *CreatePanelWithComboBox(wxChoicebook *parent);
wxPanel *CreatePanelWithCheckBox(wxChoicebook *parent);
wxPanel *CreatePanelWithListBox(wxChoicebook *parent );
wxPanel *CreatePanelWithRadioBox(wxChoicebook *parent );
wxPanel *CreatePanelWithSpinCtrl(wxChoicebook *parent );
wxPanel *CreatePanelWithTextCtrl(wxChoicebook *parent );
```

```
wxPanel *CreatePanelWithToggleButton(wxChoicebook *parent );
#endif
```

In der Quelldatei *base.cpp* findet jetzt das ganze Geschehen im Konstruktor des Frames statt:

```
// base.cpp
...
...
// Konstruktor für das Frame
BasicFrame::BasicFrame (
   const wxChar *title,
   int xpos, int ypos,
   int width, int height)
   : wxFrame ( (wxFrame *) NULL,
               -1, title,
               wxPoint(xpos, ypos),
               wxSize(width, height),wxDEFAULT_FRAME_STYLE)
{
   // Eine Menübar erzeugen
   MenuBar = new wxMenuBar();
   // Ein Menü erzeugen
   ExampleMenu = new wxMenu();
   ExampleMenu->Append(MENU_FILE_QUIT, wxT("&Beenden"));
   MenuBar->Append( ExampleMenu, wxT("&Datei"));
   SetMenuBar(MenuBar);

   // Ein choicebook erzeugen
   choicebook = new wxChoicebook(
      this, wxID_ANY, wxDefaultPosition, wxSize(300, 200),
      wxNB_MULTILINE | wxNB_FIXEDWIDTH);

   // Neue Seiten erzeugen ...
   wxPanel* window1 = CreatePanelWithButton(choicebook);
   wxPanel* window2 =
      CreatePanelWithBitmapButton(choicebook);
   wxPanel* window3 = CreatePanelWithChoice(choicebook);
   wxPanel* window4 = CreatePanelWithComboBox(choicebook);
   wxPanel* window5 = CreatePanelWithCheckBox(choicebook);
   wxPanel* window6 = CreatePanelWithListBox(choicebook);
   wxPanel* window7 = CreatePanelWithRadioBox(choicebook);
   wxPanel* window8 = CreatePanelWithSpinCtrl(choicebook);
   wxPanel* window9 = CreatePanelWithTextCtrl(choicebook);
   wxPanel* window10 =
      CreatePanelWithToggleButton(choicebook);
```

```
   // ... und zum choicebook hinzufügen
   choicebook->AddPage( window1, wxT("wxButton"), true, 0);
   choicebook->AddPage( window2,
      wxT("wxBitmapButton"), false, 1 );
   choicebook->AddPage( window3, wxT("wxChoice"), false, 2);
   choicebook->AddPage( window4,
      wxT("wxComboBox"), false, 3 );
   choicebook->AddPage( window5,
      wxT("wxCheckBox"), false, 4);
   choicebook->AddPage( window6,
      wxT("wxListBox"), false, 5 );
   choicebook->AddPage( window7,
      wxT("wxRadioBox"), false, 6 );
   choicebook->AddPage( window8,
      wxT("wxSpinCtrl"), false, 7 );
   choicebook->AddPage( window9,
      wxT("wxTextCtrl"), false, 8 );
   choicebook->AddPage( window10,
      wxT("wxToggleButton"), false,9 );
   // Eine Statusleiste für die "Ausgabe" erzeugen
   CreateStatusBar(1);
}
```

Auf den folgenden Seiten wird auf die einzelnen nicht statischen Elemente eingegangen, die im Frame mit den globalen `CreatePanelWith...`-Funktionen erzeugt werden.

wxButton

`wxButton` ist ein einfacher Button zum Drücken mit einem Text-Label. Häufig wird ein Button in einer Dialogbox, einem Panel und natürlich auch in jedem anderen Fenster platziert.

Hinweis **[«]**

Button oder *Knopf*: Auch hier bin ich der Auffassung, dass Button der geläufigere Begriff ist.

Der Konstruktor zum Erzeugen und Anzeigen eines `wxButton` sieht folgendermaßen aus:

```
wxButton( wxWindow* parent,
          wxWindowID id,
          const wxString& label = wxEmptyString,
          const wxPoint& pos = wxDefaultPosition,
          const wxSize& size = wxDefaultSize,
```

```
        long style = 0,
        const wxValidator& validator = wxDefaultValidator,
        const wxString& name = "button" );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster von `wxButton` (sollte niemals `NULL` sein)
- `id` – die Identifizierung
- `label` – der Text, der auf dem Button angezeigt wird
- `pos` – die Position des Buttons; bei (–1, –1) wird wieder die Default-Position verwendet.
- `size` – die Größe des Buttons; bei (–1, –1) wird wieder die Default-Größe verwendet.
- `style` – der Button-Stil (siehe Tabelle 10.19)
- `validator` – der Button-Validator
- `name` – der Name für den Button (wird nur unter Motif benötigt)

wxButton style	Bedeutung
`wxBU_LEFT`	links ausgerichtetes Text-Label
`wxBU_TOP`	Das Text-Label wird am oberen Teil des Buttons ausgerichtet.
`wxBU_RIGHT`	rechts ausgerichtetes Text-Label
`wxBU_BOTTOM`	Das Text-Label wird am unteren Teil des Buttons ausgerichtet.
`wxBU_EXACTFIT`	Erzeugt den Button so schmal wie möglich (anstatt die Standardgröße zu verwenden).
`wxNO_BORDER`	Erzeugt einen etwas flach wirkenden Button ohne den 3D-Rahmen.

Tabelle 10.19 »wxButton style«

Die wichtigsten Methoden von `wxButton` sind in Tabelle 10.20 aufgelistet.

Methode	Beschreibung
`wxString GetLabel() const;`	Gibt das Text-Label des Buttons zurück.
`void SetDefault();`	Setzt diesen Button als Standard-Button des Eltern-Fensters. Wird zum Beispiel ↵ gedrückt, wird dieser Button gedrückt.
`void SetLabel( const wxString& label );`	Setzt das Text-Label für den Button.

Tabelle 10.20 Wichtige Methoden von »wxButton«

`wxButton` generiert ein Ereignis der Klasse `wxCommandEvent`. Auf das Ereignis können Sie folgendermaßen mit einem Handle reagieren:

Handle einrichten	Beschreibung
`EVT_BUTTON(id, func)`	Behandelt ein `wxEVT_COMMAND_BUTTON_CLICKED`-Ereignis, das generiert wird, wenn der Anwender einen Button mit der linken Maustaste anklickt.

Tabelle 10.21 »wxButton«-Ereignisse behandeln

Abhängig von der Plattform können Sie auch beim Text-Label eines Buttons das Ampersand-Zeichen »&« verwenden. Damit markieren Sie, dass das nächste Zeichen unterstrichen wird und/oder legen eine Zugriffstaste an, die zum Einsatz kommt, wenn der Anwender die entsprechende Taste drückt. Wird dieses Feature auf Ihrer Plattform nicht unterstützt, wird das Ampersand-Zeichen automatisch entfernt und ignoriert.

Auf einigen Systemen (die GTK+ verwenden) sind außerdem auch Standard-Buttons vorhanden, die mit einer speziellen Grafik angezeigt werden. Dies sind dann typische Look-and-Feel-Grafiken wie beispielsweise die Schere fürs Ausschneiden oder die Diskette fürs Speichern.

Wenn Sie diese Funktion verwenden wollen, gehen Sie zum Beispiel folgendermaßen vor:

```
wxButton* button = new wxButton(this, wxID_OK);
```

Ein Text-Label ist hierfür nicht nötig, weil `wxWidgets` für das korrekte Label sorgt (auf allen Plattformen). So wird beispielsweise unter MS Windows und Mac OS X die Zeichenkette `"&OK"` eingesetzt. Unter GTK+ wird der entsprechende Button mit einem Icon verwendet (auch *Stock-Button* genannt).

Natürlich können Sie trotzdem den Standard-Identifizierer `wxID_OK` mit einem Text-Label verwenden:

```
wxButton* button =
   new wxButton(this, wxID_OK, "&OKIDOKI");
```

Jetzt befindet sich `"OKIDOKI"` auf dem Label (auf allen Plattformen).

In der folgenden Tabelle (10.22) finden Sie einige dieser Stock-Button-Identifizierer (zweiter Parameter), die Sie verwenden können (unter GTK+ sehen Sie auch das Icon).

Hinweis **[«]**

Man sollte beachten, dass durch die Verwendung der Stock-Buttons mit `wxID_...` die Labels in der Regel in englischer Sprache angezeigt werden.

Stock-Button-ID	Stock-Button-Label
wxID_ADD	»Add«
wxID_APPLY	»&Apply«
wxID_BOLD	»&Bold«
wxID_CANCEL	»&Cancel«
wxID_CLEAR	»&Clear«
wxID_CLOSE	»&Close«
wxID_COPY	»&Copy«
wxID_CUT	»Cu&t«
wxID_DELETE	»&Delete«
wxID_FIND	»&Find«
wxID_REPLACE	»Rep&lace«
wxID_BACKWARD	»&Back«
wxID_DOWN	»&Down«
wxID_FORWARD	»&Forward«
wxID_UP	»&Up«
wxID_HELP	»&Help«
wxID_HOME	»&Home«
wxID_INDENT	»Indent«
wxID_INDEX	»&Index«
wxID_ITALIC	»&Italic«
wxID_JUSTIFY_CENTER	»Centered«
wxID_JUSTIFY_FILL	»Justified«
wxID_JUSTIFY_LEFT	»Align Left«
wxID_JUSTIFY_RIGHT	»Align Right«
wxID_NEW	»&New«
wxID_NO	»&No«
wxID_OK	»&OK«
wxID_OPEN	»&Open«
wxID_PASTE	»&Paste«
wxID_PREFERENCES	»&Preferences«
wxID_PRINT	»&Print«
wxID_PREVIEW	»Print previe&w«
wxID_PROPERTIES	»&Properties«
wxID_EXIT	»&Quit«
wxID_REDO	»&Redo«

Tabelle 10.22 Stock-Button-Identifizierer

Stock-Button-ID	Stock-Button-Label
wxID_REFRESH	»Refresh«
wxID_REMOVE	»Remove«
wxID_REVERT_TO_SAVED	»Revert to Saved«
wxID_SAVE	»&Save«
wxID_SAVEAS	»Save &As...«
wxID_STOP	»&Stop«
wxID_UNDELETE	»Undelete«
wxID_UNDERLINE	»&Underline«
wxID_UNDO	»&Undo«
wxID_UNINDENT	»&Unindent«
wxID_YES	»&Yes«
wxID_ZOOM_100	»&Actual Size«
wxID_ZOOM_FIT	»Zoom to &Fit«
wxID_ZOOM_IN	»Zoom &In«
wxID_ZOOM_OUT	»Zoom &Out«

Tabelle 10.22 Stock-Button-Identifizierer (Forts.)

Im Beispiel *base.cpp* erzeugen wir im Konstruktor `BasicFrame` folgendermaßen ein Panel mit `wxButton`:

```
wxPanel* window1 = CreatePanelWithButton(choicebook);
```

Die globale Funktion `CreatePanelWithButton`, in der die Buttons der Klasse `wxButton` erzeugt werden, sieht so aus:

```
wxPanel *CreatePanelWithButton(wxChoicebook *parent) {
   wxPanel *panel = new wxPanel(parent, wxID_ANY);
   // Text-Label zum Panel
   (void )new wxStaticText(panel, wxID_STATIC,
           wxT("Demonstriert Buttons der Klasse wxButton"),
           wxPoint(10, -1), wxDefaultSize, wxALIGN_LEFT);
   // Eine Menge Buttons für das Panel
   (void) new wxButton(
         panel, ID_BUTTON1, wxT("Button1"),
         wxPoint(10,30), wxDefaultSize);
   (void) new wxButton(
         panel, wxID_OK, wxT("Ok"), wxPoint( 10, 60 ) );
   (void) new wxButton(
         panel, wxID_CANCEL, wxT("Cancel"), wxPoint(10,90));
   (void) new wxButton(
         panel, wxID_ANY, wxT("wxBU_LEFT"),
```

```
          wxPoint( 100, 30 ), wxDefaultSize, wxBU_LEFT );
   (void) new wxButton(
          panel, wxID_ANY, wxT("wxNO_BORDER"),
          wxPoint( 100, 60), wxDefaultSize, wxNO_BORDER );
   (void) new wxButton(
          panel, wxID_ANY, wxT("wxBU_EXACTFIT"),
          wxPoint(100, 90), wxDefaultSize, wxBU_EXACTFIT  );
   (void) new wxButton(
          panel, wxID_ANY, wxT("wxSize(100, 90)"),
          wxPoint( 230, 30 ), wxSize(100, 90) );
    return panel;
}
```

Dieses Panel fügen wir nun zu unserem *Choicebook* hinzu

```
choicebook->AddPage(window1, wxT("wxButton"), true, 0);
```

und erhalten folgendes Bild bei der Ausführung:

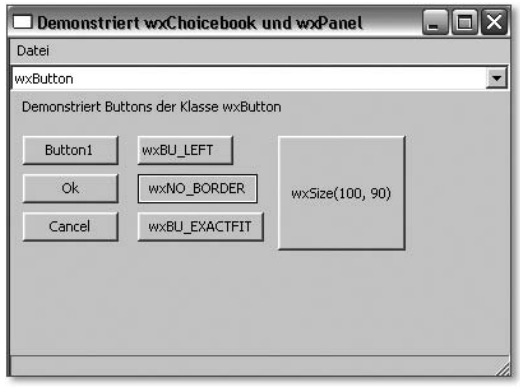

Abbildung 10.22 »wxButton« im Einsatz

Natürlich soll hierbei auch auf die Ereignisse reagiert werden. Die entsprechenden Ereignis-Handle richtet man folgendermaßen ein (im Beispiel wird nur auf die ersten drei der sieben Buttons reagiert):

```
// Ereignis-Tabelle für das Frame
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   ...
   EVT_BUTTON(ID_BUTTON1, BasicFrame::OnButton1)
   EVT_BUTTON(wxID_OK, BasicFrame::OnButtonOk)
   EVT_BUTTON(wxID_CANCEL, BasicFrame::OnButtonCancel)
   ...
END_EVENT_TABLE()
```

Im Ereignis-Handle stellen wir mit der Methode `wxWindow::FindWindowByID` erst einmal einen Bezug zu den entsprechenden Buttons her. Die Methode `FindWindowByID` erhält dabei den Identifizierer (zweiter Parameter von `wxButton`) als Parameter. Neben dieser Methode sind in `wxWindow` mit `FindWindow`, `FindWindowByName` und `FindWindowByLabel` noch drei weitere ähnliche Methoden definiert, die in Tabelle 10.23 näher beschrieben sind. Da der Rückgabewert hierbei vom Typ `wxWindow` ist, müssen Sie ihn in die entsprechende Klasse umwandeln.

Methode	Beschreibung
`wxWindow* FindWindow( long id ) const;`	Sucht nach einem Fenster oder Element mit dem Identifizierer `id`.
`wxWindow* FindWindow( const wxString& name ) const;`	Sucht nach einem Fenster oder Element mit dem Namen `name`, das als letzter Parameter bei den Konstruktoren angegeben werden kann.
`static wxWindow* FindWindowById( long id, wxWindow* parent = NULL );`	Sucht nach dem ersten Fenster oder Element mit dem Identifizierer `id`. Wird hier für `parent` `NULL` angegeben, wird mit der Suche vom Top-Level-Frame angefangen. Ansonsten können Sie hierbei angeben, ab welcher Ebene die Methode anfangen soll zu suchen. Beachten Sie allerdings, dass nichts gefunden wird, wenn sich `id` über dieser Ebene befindet.
`static wxWindow* FindWindowByLabel( const wxString& label, wxWindow* parent = NULL );`	Sucht nach einem Fenster oder Element mit dem Text `label`. Bei einem Fenster ist dies beispielsweise der Titel, bei einem Button das Text-Label (dies ist vom Element abhängig). Wird hier für `parent` `NULL` angegeben, wird mit der Suche vom Top-Level-Frame angefangen. Ansonsten können Sie hierbei angeben, ab welcher Ebene die Methode anfangen soll zu suchen. Beachten Sie allerdings, dass nichts gefunden wird, wenn sich `id` über dieser Ebene befindet.
`static wxWindow* FindWindowByName( const wxString& name, wxWindow* parent = NULL );`	Sucht nach einem Fenster oder Element mit dem Namen `name`, das als letzter Parameter bei den Konstruktoren angegeben werden kann. Wird hier für `parent` `NULL` angegeben, wird mit der Suche vom Top-Level-Frame angefangen. Ansonsten können Sie hierbei angeben, ab welcher Ebene die Methode anfangen soll zu suchen. Beachten Sie allerdings, dass nichts gefunden wird, wenn sich `id` über dieser Ebene befindet. Wird kein Fenster oder Element mit solch einem Namen gefunden, wird auch noch die Methode `FindWindowByLabel` aufgerufen.

Tabelle 10.23 Methoden von »wxWindow« zum Suchen nach Widgets

Hierzu noch die entsprechenden Ereignis-Handles, die für `wxButton` eingerichtet wurden:

```
// ... wenn ein wxButton betätigt wurde
void BasicFrame::OnButton1(wxCommandEvent &event) {
   wxButton *but = (wxButton*) FindWindowById(ID_BUTTON1);
   wxString s;
   s.Append(but->GetLabel());
   s.Append(wxT(" wurde gedrückt"));
   SetStatusText(s);
}

void BasicFrame::OnButtonOk(wxCommandEvent &event) {
   wxButton *but = (wxButton*) FindWindowById(wxID_OK);
   wxString s;
   s.Append(but->GetLabel());
   s.Append(wxT(" wurde gedrückt"));
   SetStatusText(s);
}

void BasicFrame::OnButtonCancel(wxCommandEvent &event) {
   wxButton *but = (wxButton*) FindWindowById(wxID_CANCEL);
   wxString s;
   s.Append(but->GetLabel());
   s.Append(wxT(" wurde gedrückt"));
   SetStatusText(s);
}
```

wxBitmapButton

Ein `wxBitmapButton` entspricht in etwa der Klasse `wxButton`, nur dass hierbei eine Bitmap-Grafik anstelle eines Textes angezeigt wird. Der Konstruktor zum Erzeugen eines `wxBitmapButton` sieht so aus:

```
wxBitmapButton(
            wxWindow* parent,
            wxWindowID id,
            const wxBitmap& bitmap,
            const wxPoint& pos = wxDefaultPosition,
            const wxSize& size = wxDefaultSize,
            long style = wxBU_AUTODRAW,
            const wxValidator& validator=wxDefaultValidator,
            const wxString& name = "button" );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster von `wxButton` (sollte niemals `NULL` sein)
- `id` – die Identifizierung
- `bitmap` – die Bitmap, das angezeigt werden soll
- `pos` – die Position des Buttons; bei (–1, –1) wird wieder die Default-Position verwendet.
- `size` – die Größe des Buttons; bei (–1, –1) wird wieder die Default-Größe verwendet.
- `style` – der Button-Stil (siehe Tabelle 10.24)
- `validator` – der Fenster-Validator
- `name` – der Name für den Button (wird nur unter Motif benötigt)

wxBitmapButton style	Bedeutung
`wxBU_AUTODRAW`	Mit diesem Flag wird ein Button mit 3D-Rahmen gezeichnet. Ohne Flag lässt sich schwer erkennen, ob es sich hier um einen Button handelt oder nur um eine Bitmap.
`wxBU_LEFT`	Bitmap ist links ausgerichtet.
`wxBU_TOP`	Bitmap wird oben am Button ausgerichtet.
`wxBU_RIGHT`	Bitmap ist rechts ausgerichtet.
`wxBU_BOTTOM`	Bitmap wird unten am Button ausgerichtet.

Tabelle 10.24 »wxBitmapButton style«

Bei den Ereignissen für `wxBitmapButton` gilt dasselbe wie für `wxButton`. Auch `wxBitmapButton` generiert ein Ereignis der Klasse `wxCommandEvent`, worauf mit einem Handle folgendermaßen reagiert werden kann:

Handle einrichten	Beschreibung
`EVT_BUTTON(id, func)`	Behandelt ein `wxEVT_COMMAND_BUTTON_CLICKED`-Ereignis, das generiert wird, wenn der Anwender einen Button mit der linken Maustaste anklickt.

Tabelle 10.25 »wxBitmapWindow«-Ereignisse behandeln

`wxBitmapButton` hat ebenfalls einige Methoden, von denen die beiden Methoden `SetBitmapLabel` und `GetBitmapLabel` zum Setzen bzw. Abfragen der Bitmap auf dem Button wohl die wichtigsten sind. Es ist aber auch möglich, mit Hilfe von `SetBitmapFocus`, `SetBitmapSelected` und `SetBitmapDisabled` die Kontrolle auf diesen Buttons zu verfeinern. Aber im Grunde werden diese Methoden recht selten eingesetzt, weshalb nicht näher darauf eingegangen wird.

Im Beispiel *base.cpp* erzeugen wir im Konstruktor `BasicFrame` folgendermaßen ein Panel mit `wxBitmapButton`:

```
wxPanel* window2 = CreatePanelWithBitmapButton(choicebook);
```

Die globale Funktion `CreatePanelWithBitmapButton`, in der die `wxBitmapButton` erzeugt werden, sieht wie folgt aus:

```
wxPanel *CreatePanelWithBitmapButton(wxChoicebook *parent) {
   wxPanel *panel = new wxPanel(parent, wxID_ANY);
   // Ein Text-Label zum Panel
   (void) new wxStaticText(panel, wxID_STATIC,
      wxT("Demonstriert Buttons der Klasse wxBitmapButton"),
      wxPoint(10, -1), wxDefaultSize, wxALIGN_LEFT);

   // Die Bitmap-Buttons für das Panel
   wxBitmap bitmap1(wxT("Smile.bmp"), wxBITMAP_TYPE_BMP );
   (void) new wxBitmapButton(
         panel, ID_BUTTON1, bitmap1,
         wxPoint(10, 20), wxDefaultSize, wxBU_AUTODRAW);

   wxBitmap bitmap2( wxT("Communication.bmp"),
                     wxBITMAP_TYPE_BMP );
   (void) new wxBitmapButton(
         panel, wxID_OK, bitmap2,
         wxPoint(50, 20), wxDefaultSize, wxBU_AUTODRAW);

   wxBitmap bitmap3(wxT("Book.bmp"), wxBITMAP_TYPE_BMP );
   (void) new wxBitmapButton(
         panel, wxID_CANCEL, bitmap3,
         wxPoint(90, 20), wxDefaultSize, wxBU_AUTODRAW);

   wxBitmap bitmap4(wxT("John.bmp"), wxBITMAP_TYPE_BMP );
   (void) new wxBitmapButton(
         panel, wxID_ANY, bitmap4,
         wxPoint(130, 20), wxDefaultSize, wxBU_AUTODRAW );
    return panel;
}
```

Dieses Panel fügen wir wieder zum *Choicebook* hinzu

```
choicebook->AddPage( window2,
                     wxT("wxBitmapButton"), false, 1 );
```

und erhalten anschließend folgendes Bild bei der Ausführung (siehe Abbildung 10.23):

Abbildung 10.23 »wxBitmapButton« im Einsatz

Auf Ereignisse reagieren wir hier nicht gesondert und verwenden dieselben Ereignisse und Handles wie bei `wxButton`.

wxChoice

`wxChoice` ist eine einfache Dropdown-Listbox, die nur gelesen werden kann. Die Liste der Box wird nicht angezeigt, bis der Anwender mit dem Mausbutton die Liste aktiviert.

`wxChoice` hat zwei Konstruktoren (abgesehen vom Default-Konstruktor), mit denen ein Element der Klasse `wxChoice` erzeugt werden kann. Die Syntax unterscheidet sich nur darin, dass eine Version einen `wxString`-Vektor enthält, bei dem die Anzahl der Elemente angegeben werden muss. Die zweite Version hat stattdessen ein `wxStringArray`, bei dem keine Angabe zur Anzahl der Elemente gemacht wird.

```
wxChoice( wxWindow *parent,
          wxWindowID id,
          const wxPoint& pos,
          const wxSize& size,
          int n,
          const wxString choices[],
          long style = 0,
          const wxValidator& validator = wxDefaultValidator,
          const wxString& name = "choice" );

wxChoice( wxWindow *parent,
          wxWindowID id,
          const wxPoint& pos,
          const wxSize& size,
```

```
        const wxArrayString& choices,
        long style = 0,
        const wxValidator& validator = wxDefaultValidator,
        const wxString& name = "choice" );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster von `wxChoice` (sollte niemals `NULL` sein)
- `id` – die Identifizierung
- `pos` – die Position; bei (–1, –1) wird wieder die Default-Position verwendet.
- `size` – die Größe; bei (–1, –1) wird wieder die Default-Größe verwendet.
- Liste der Strings (Version 1):
  - `n` – Anzahl der Strings in der Liste
  - `choices` – ein Array mit Strings (`wxString`), die angezeigt werden sollen
- Liste der Strings (Version 2):
  - `choices` – ein Array mit Strings (`wxStringArray`), die angezeigt werden sollen
- `style` – der Button-Stil (`wxChoice` hat hierbei keine speziellen Styles)
- `validator` – der Validator
- `name` – der Name für `wxChoice`

Zwar hat `wxChoice` auch einige Methoden, aber in der Praxis wird auf die Methoden der abstrakten Basisklasse `wxControlWithItems` zurückgegriffen. Da auch die Klassen `wxListBox`, `wxCheckListBox` und `wxComboBox` davon abgeleitet sind, soll auf diese Methoden in der folgenden Tabelle (10.26) etwas näher eingegangen werden.

Methode	Beschreibung
`int Append( const wxString& item );`	Fügt den neuen String am Ende der Liste ein.
`int Append( const wxString& item, void *clientData );`	Fügt den neuen String am Ende der Liste ein und übergibt dazu einen Zeiger auf die Client-Daten.
`void Append( const wxArrayString& strings);`	Fügt mehrere Strings auf einmal am Ende der Liste ein.
`void Clear();`	Entfernt alle Elemente einer Liste.
`void Delete(int n);`	Entfernt das Element `n` (angefangen bei `0`) von der Liste.

Tabelle 10.26 Methoden von »wxControlWithItems«

Methode	Beschreibung
`int FindString( const wxString& string, bool caseSensitive = false );`	Findet ein Element, das mit dem Text-Label `string` übereinstimmt. Außerdem können Sie mit dem zweiten Parameter einstellen, ob die Suche case-sensitiv (Unterscheidung zwischen Groß- und Kleinbuchstaben) sein soll.
`void * GetClientData( int n ) const;`	Gibt einen untypisierten Zeiger auf die Client-Daten zurück, die mit dem Listenelement verknüpft sind (wenn eins vorhanden ist). Mit `n` geben Sie die gewünschte Position des Elements in der Liste an.
`wxClientData * GetClientObject( int n ) const;`	Dito, wie `GetClientData`, nur ist der Zeiger typisiert (`wxClientData*`).
`int GetCount() const;`	Gibt die Anzahl der Einträge der Liste zurück.
`int GetSelection() const;`	Gibt den Index des ausgewählten Elements der Liste zurück. Diese Methode kann allerdings nur in Verbindung mit Listen verwendet werden, mit denen einzelne Elemente ausgewählt werden. Verwenden Sie beispielsweise `wxListBox`, wo mit der Option `wxLB_MULTIPLE` mehrere Elemente ausgewählt werden können, müssen Sie die entsprechende `wxListBox::GetSelections` verwenden.
`wxString GetString(int n) const;`	Gibt den String zurück, den der Listeneintrag mit dem Index `n` enthält.
`wxString GetStringSelection() const;`	Gibt den String des ausgewählten Elements der Liste zurück.

`int Insert( const wxString& item, int pos );`
`int Insert( const wxString& item, int pos, void *clientData);`
`int Insert( const wxString& item, int pos, wxClientData *clientDat);`
	Wie `Append`, nur kann hiermit ein Element an der Position `pos` eingefügt werden. Beachten Sie, dass diese Methode nicht mit den Flags `wxLB_SORT` oder `wxCB_SORT` funktioniert. Ist die Liste sortiert, sollten Sie stattdessen die Methode `Append` verwenden.

`bool IsEmpty() const;`	Gibt `true` zurück, wenn die Liste leer ist. `false` wird zurückgegeben, wenn die Liste nicht leer ist.
`void Select(int n);`	Wie `SetSelection`, nur können hiermit mehrere Elemente gleichzeitig ausgewählt werden (wenn es das Widget erlaubt).

Tabelle 10.26 Methoden von »wxControlWithItems« (Forts.)

Methode	Beschreibung
`void SetClientData( int n, void *data);`	Verknüpft das Listenelement `n` mit untypisierten Daten.
`void SetClientObject( int n, wxClientData *data );`	Wie `SetClientData`, nur sind die Daten typisiert (`wxClientData*`).
`void SetSelection(int n);`	Markiert das Listenelement `n` als ausgewählt. Es wird aber kein Ereignis ausgelöst.
`void SetString( int n, const wxString& string );`	Setzt den String für das Listenelement `n` mit `string`.
`bool SetStringSelection( const wxString& string );`	Markiert das Listenelement mit `string` als ausgewählt. Hierbei wird auch kein Ereignis ausgelöst.

Tabelle 10.26 Methoden von »wxControlWithItems« (Forts.)

[»]

Hinweis

Neben dem String-Label, den jedes `wxControlWithItems` besitzt, kann man optional auch Client-Daten mit diesem verknüpfen. Dabei kann man entweder untypisierte Daten (`void*`), mit denen man einfache Daten anhängen kann, oder typisierte Daten (`wxClientData*`) anhängen. Wie dies in der Praxis funktioniert, werden Sie anschließend im Listing sehen.

`wxChoice` generiert ein Ereignis der Klasse `wxCommandEvent`. Auf das Ereignis können Sie mit einem Handle folgendermaßen reagieren:

Handle einrichten	Beschreibung
`EVT_CHOICE(id, func)`	Behandelt ein `wxEVT_COMMAND_CHOICE_SELECTED`-Ereignis, das generiert wird, wenn der Anwender ein Element aus der Liste auswählt.

Tabelle 10.27 »wxChoice«-Ereignisse behandeln

Im Beispiel *base.cpp* erzeugen wir im Konstruktor `BasicFrame` folgendermaßen ein Panel mit `wxChoice`:

```
wxPanel* window3 = CreatePanelWithChoice(choicebook);
```

Die globale Funktion `CreatePanelWithChoice`, in der `wxChoice` erzeugt wird, sieht wie folgt aus:

```
wxPanel *CreatePanelWithChoice(wxChoicebook *parent) {
   wxPanel *panel = new wxPanel(parent, wxID_ANY);
   wxArrayString obst;
```

```
    obst.Add(wxT("Birne"));
    obst.Add(wxT("Orange"));
    obst.Add(wxT("Apfel"));
    obst.Add(wxT("Kirsche"));

    wxArrayString gemuese;
    gemuese.Add(wxT("Tomate"));
    gemuese.Add(wxT("Zwiebel"));
    gemuese.Add(wxT("Lauch"));
    gemuese.Add(wxT("Kartoffel"));

    // Text-Label zum Panel hinzufügen
    (void) new wxStaticText(panel, wxID_STATIC,
      wxT("Demonstriert die Listen-Box der Klasse wxChoice"),
      wxPoint(10, -1), wxDefaultSize, wxALIGN_LEFT );

    // Die Liste von Strings erzeugen
    wxChoice *m_choice = new wxChoice(
          panel, ID_CHOICE,
          wxPoint(10, 20), wxSize(120,-1), obst );
    // "Birne" bekommt noch spezielle Client-Daten
    m_choice->SetClientData(
          0, (void*) wxT("Birnen schmecken gut"));
    // ... die "Orangen" auch
    m_choice->SetClientData(1, (void*)wxT("Viel Vitamin C"));

    // Jetzt werden die einzelnen Strings ABC sortiert
    (void) new wxChoice(
          panel, ID_CHOICE_SORTED,
          wxPoint(10, 80), wxSize(120,-1),
          obst, wxCB_SORT );
    // Noch eine sortierte Liste erstellen
    (void) new wxChoice(
          panel, ID_CHOICE_SORTED2,
          wxPoint(220,20), wxSize(120,-1),
          gemuese, wxCB_SORT );
    // Einen Button zum Auswerten hinzufügen
    (void) new wxButton(
          panel, ID_CHOICE_ALL, wxT("Alle Auswerten"),
          wxPoint(220,80), wxSize(120,-1) );
    return panel;
}
```

Jetzt wird das Panel nur noch zum *Choicebook* hinzugefügt

```
choicebook->AddPage(window3, wxT("wxChoice"), false, 2);
```

und Sie erhalten anschließend folgendes Bild bei der Ausführung:

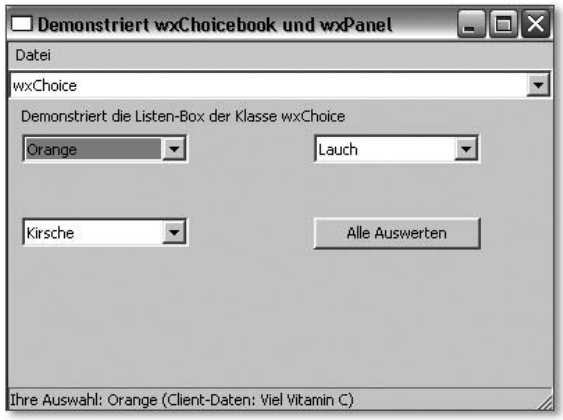

Abbildung 10.24 »wxChoice« bei der Ausführung

Hierfür wollen wir selbstverständlich auch wieder die Ereignis-Handles einrichten:

```
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   ...
   EVT_CHOICE(ID_CHOICE, BasicFrame::OnChoice1)
   EVT_CHOICE(ID_CHOICE_SORTED, BasicFrame::OnChoice1)
   EVT_CHOICE(ID_CHOICE_SORTED2, BasicFrame::OnChoice1)
   EVT_BUTTON(ID_CHOICE_ALL, BasicFrame::OnChoice2)
   ...
END_EVENT_TABLE()
```

Neben der Möglichkeit, einzelne Dropdown-Listen auszuwerten (`OnChoice1`), wurde auch ein Ereignis-Handle für den Button hinzugefügt (`OnChoice2`), in dem der aktuelle Zustand aller Dropdown-Listen ausgewertet werden kann. Beim Ereignis-Handle `OnChoice1` wurde außerdem die Methode `wxEvent::GetEvent-Object` verwendet, um das Objekt zu ermitteln, das das Ereignis ausgelöst hat. Hier die entsprechenden Ereignis-Handles dazu:

```
// Ein Listenelement wurde ausgewählt ...
void BasicFrame::OnChoice1( wxCommandEvent &event ) {
   wxChoice *choice = (wxChoice*) event.GetEventObject();
   wxString s(wxT("Ihre Auswahl: "));
   s.Append(choice->GetStringSelection());
   // ... sind auch Client-Daten vorhanden?
   if( choice->GetClientData(choice->GetSelection()) ) {
```

```
      s.Append(wxT(" (Client-Daten: "));
      s.Append(
         (const wxChar*) choice->GetClientData(
             choice->GetSelection() ) );
      s.Append(wxT(")"));
   }
   SetStatusText( s );
}

// Alle Listenelemente auswerten ...
void BasicFrame::OnChoice2( wxCommandEvent &event ) {
   wxChoice *choice1 = (wxChoice*)
      FindWindowById(ID_CHOICE);
   wxChoice *choice2 = (wxChoice*)
      FindWindowById(ID_CHOICE_SORTED);
   wxChoice *choice3 = (wxChoice*)
      FindWindowById(ID_CHOICE_SORTED2);

   wxString s(wxT("Gesamte Auswahl: "));
   s.Append(choice1->GetStringSelection());
   s.Append(wxT(" & "));
   s.Append(choice2->GetStringSelection());
   s.Append(wxT(" & "));
   s.Append(choice3->GetStringSelection());
   // Ergebnis in die Statuszeile
   SetStatusText( s );
}
```

wxComboBox

Die Combo-Box der Klasse `wxComboBox` ist der Klasse von `wxChoice` recht ähnlich. Nur ist diese Box eine Mischung aus Dropdown-Liste (wie gehabt) und einem einzeiligen Textfeld, das heißt, Sie können auch einen neuen Text hinzufügen.

Die Syntax des Konstruktors von `wxComboBox` ist, wie schon bei `wxChoice`, in zweifacher Ausführung vorhanden:

```
wxComboBox( wxWindow* parent,
            wxWindowID id,
            const wxString& value = "",
            const wxPoint& pos = wxDefaultPosition,
            const wxSize& size = wxDefaultSize,
            int n,
            const wxString choices[],
            long style = 0,
            const wxValidator& validator=wxDefaultValidator,
            const wxString& name = "comboBox" );
```

```
wxComboBox( wxWindow* parent,
            wxWindowID id,
            const wxString& value,
            const wxPoint& pos,
            const wxSize& size,
            const wxArrayString& choices,
            long style = 0,
            const wxValidator& validator=wxDefaultValidator,
            const wxString& name = "comboBox" );
```

Auch die Bedeutung der einzelnen Parameter können Sie der Klasse `wxChoice` entnehmen – abgesehen davon, dass `wxComboBox` verschiedene Stile anbietet (siehe Tabelle 10.28) und dass Sie mit `value` einen String angeben können, der beim Programmstart ausgewählt (selektiert) wird. Der String muss natürlich im String-Array `choices` vorhanden sein.

wxComboBox Style	Beschreibung
`wxCB_SIMPLE`	Erzeugt eine Combo-Box, in der die komplette Liste permanent angezeigt wird.
`wxCB_DROPDOWN`	Erzeugt eine Combo-Box mit einer Dropdown-Liste (Standardeinstellung.)
`wxCB_READONLY`	Erzeugt eine Combo-Box, in der zwar Elemente der Liste ausgewählt werden können, aber das Textfeld nicht verwendet werden kann. Entspricht dann der Klasse `wxChoice`.
`wxCB_SORT`	Erzeugt eine Combo-Box, in der die Liste immer alphabetisch sortiert ist.

Tabelle 10.28 »wxComboBox«-Stile

Auch `wxComboBox` bietet einige interessante Methoden an, doch auch hier wird man wohl vorwiegend von den Methoden der Klasse `wxControlWithItems` (siehe Tabelle 10.29) Gebrauch machen. Trotzdem seien einige Methoden von `wxComboBox` hier erwähnt (siehe ebenfalls Tabelle 10.29). Vorwiegend handelt es sich dabei um Methoden, die sich auf das editierbare Textfeld der Combo-Box beziehen. Sofern Sie beim Erzeugen von `wxComboBox` `wxCB_READONLY` verwendet haben, lassen sich die schreibenden Methoden natürlich nicht darauf anwenden.

Methode	Beschreibung
`void Copy();`	Kopiert den ausgewählten Text in die Zwischenablage.
`void Cut();`	Kopiert den ausgewählten Text in die Zwischenablage und entfernt die Auswahl (allerdings nicht aus der Listbox, sondern nur aus dem Editierfeld).

Tabelle 10.29 Methoden von »wxComboBox«

Methode	Beschreibung
`long GetInsertionPoint() const;`	Gibt den Einfügepunkt für das Textfeld in der Combo-Box zurück.
`virtual wxTextPos GetLastPosition() const,`	Gibt die letzte Position des Textfelds der Combo-Box zurück.
`wxString GetValue() const;`	Gibt den aktuellen Wert im Textfeld zurück.
`void Paste();`	Fügt Text aus der Zwischenablage zum Textfeld hinzu.
`void Replace( long from, long to, const wxString& text );`	Ersetzt den Text von der Position `from` bis zur Position `to` durch den Text `text`.
`void Remove(long from, long to);`	Entfernt den Text zwischen der Position `from` und der Position `to`.
`void SetInsertionPoint( long pos );`	Setzt den Einfügepunkt im Textfeld auf die Position `pos`.
`void SetInsertionPointEnd();`	Setzt den Einfügepunkt auf das Ende des Textfelds.
`void SetSelection( long from, long to );`	Selektiert den Text im Textfeld ab der Position `from` bis zur Position `to`.
`void SetValue( const wxString& text );`	Setzt den Text im Textfeld auf `text` (dies ersetzt aber nicht den Original-String oder fügt den Text in der Combo-Box hinzu).

Tabelle 10.29 Methoden von »wxComboBox« (Forts.)

`wxComboBox` generiert ein Ereignis der Klasse `wxCommandEvent`. Auf das Ereignis können Sie mit einem Handle folgendermaßen reagieren:

Handle einrichten	Beschreibung
`EVT_TEXT(id, func)`	Behandelt ein `wxEVT_COMMAND_TEXT_UPDATED`-Ereignis, das generiert wird, wenn der Anwender das Textfeld editiert.
`EVT_COMBOBOX(id, func)`	Behandelt ein `wxEVT_COMMAND_COMBOBOX_SELECTED`-Ereignis, das generiert wird, wenn der Anwender ein Element aus der Liste auswählt.

Tabelle 10.30 »wxComboBox«-Ereignisse behandeln

In *base.cpp* erzeugen wir im Konstruktor `BasicFrame` wie folgt ein Panel mit `wxComboBox`:

```
wxPanel* window4 = CreatePanelWithComboBox(choicebook);
```

Die globale Funktion `CreatePanelWithComboBox`, in der `wxComboBox` erzeugt wird, sieht so aus:

```
wxPanel *CreatePanelWithComboBox(wxChoicebook *parent) {
   wxPanel *panel = new wxPanel(parent, wxID_ANY);
   wxArrayString obst;
   obst.Add(wxT("Birne"));
   obst.Add(wxT("Orange"));
   obst.Add(wxT("Apfel"));
   obst.Add(wxT("Kirsche"));

   wxArrayString gemuese;
   gemuese.Add(wxT("Tomate"));
   gemuese.Add(wxT("Zwiebel"));
   gemuese.Add(wxT("Lauch"));
   gemuese.Add(wxT("Kartoffel"));

   (void) new wxStaticText( panel, wxID_STATIC,
      wxT("Demonstriert Combo-Boxen der Klasse wxComboBox"),
      wxPoint(10, -1), wxDefaultSize, wxALIGN_LEFT );

   // Die Liste von Strings erzeugen
   (void) new wxComboBox (
         panel, ID_COMBO, wxT("Orange"),
         wxPoint(10, 20), wxSize(120,-1),
         obst, wxCB_DROPDOWN );
   // Jetzt werden die einzelnen Strings nach ABC sortiert
   (void) new wxComboBox (
         panel, ID_COMBO_SORTED, wxT("Apfel"),
         wxPoint(10, 80 ), wxSize(120,-1),
         obst, wxCB_SORT );
   // Noch eine sortierte Liste erstellen
   (void) new wxComboBox (
         panel, ID_COMBO_SORTED2, wxT("Zwiebel"),
         wxPoint(220, 20 ), wxSize(120,-1),
         gemuese, wxCB_SORT );
   // Einen Button zum Auswerten hinzufügen
   (void) new wxButton(
         panel, ID_COMBO_ALL, wxT("Alle Auswerten"),
         wxPoint(220, 80), wxSize(120,-1) );
   // Einen Button für Elemente hinzufügen
   (void) new wxButton(
         panel, ID_COMBO_ADD, wxT("Hinzufügen"),
         wxPoint( 220, 110), wxSize(120, -1) );
   return panel;
}
```

Wieder zum *Choicebook* hinzugefügt

```
choicebook->AddPage(window4, wxT("wxComboBox"), false, 3);
```

erhalten Sie folgendes Bild bei der Ausführung:

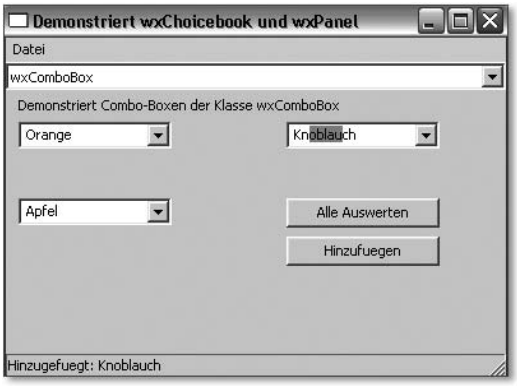

Abbildung 10.25 »wxComboBox« im Einsatz

Den Ereignis-Handle richten wir wie folgt ein:

```
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   ...
   EVT_COMBOBOX (ID_COMBO, BasicFrame::OnCombo1)
   EVT_COMBOBOX (ID_COMBO_SORTED, BasicFrame::OnCombo1)
   EVT_COMBOBOX (ID_COMBO_SORTED2, BasicFrame::OnCombo1)
   EVT_BUTTON(ID_COMBO_ALL, BasicFrame::OnCombo2)
   EVT_BUTTON(ID_COMBO_ADD, BasicFrame::OnCombo3)
   EVT_TEXT(ID_COMBO, BasicFrame::ComboText)
   ...
END_EVENT_TABLE()
```

In den darauffolgenden Ereignis-Handles finden Sie auch keine Neuerungen mehr:

```
// Ein Element in der Combo-Box wurde ausgewählt ...
void BasicFrame::OnCombo1( wxCommandEvent &event ) {
   wxComboBox *combo = (wxComboBox*) event.GetEventObject();
   wxString s(wxT("Ihre Auswahl: "));
   s.Append(combo->GetValue());
   SetStatusText( s );
   combo->Cut();
}

// Alle Combos auswerten
```

```
void BasicFrame::OnCombo2( wxCommandEvent &event ) {
   wxComboBox *combo1 = (wxComboBox*)
      FindWindowById(ID_COMBO);
   wxComboBox *combo2 = (wxComboBox*)
      FindWindowById(ID_COMBO_SORTED);
   wxComboBox *combo3 = (wxComboBox*)
      FindWindowById(ID_COMBO_SORTED2);

   wxString s(wxT(""));
   s.Append(combo1->GetValue());
   s.Append(wxT(" & "));
   s.Append(combo2->GetValue());
   s.Append(wxT(" & "));
   s.Append(combo3->GetValue());
   SetStatusText( s );
}

// Neue Elemente zur Combo-Box hinzufügen
void BasicFrame::OnCombo3( wxCommandEvent &event ) {
   wxComboBox *combo1 = (wxComboBox*)
      FindWindowById(ID_COMBO);
   wxComboBox *combo2 = (wxComboBox*)
      FindWindowById(ID_COMBO_SORTED);
   wxComboBox *combo3 = (wxComboBox*)
      FindWindowById(ID_COMBO_SORTED2);
   wxString s = wxT("Hinzugefügt: ");

   if(combo1->FindString(combo1->GetValue())==wxNOT_FOUND) {
      combo1->Append( combo1->GetValue());
      s.Append(combo1->GetValue());
      s.Append(wxT(" "));
   }
   if(combo2->FindString(combo2->GetValue())==wxNOT_FOUND) {
      combo2->Append( combo2->GetValue());
      s.Append(combo2->GetValue());
      s.Append(wxT(" "));
   }
   if(combo3->FindString(combo3->GetValue())==wxNOT_FOUND) {
      combo3->Append( combo3->GetValue());
      s.Append(combo3->GetValue());
   }
   SetStatusText( s );
}

// Ein Element wurde im Textfeld der Combo-Box verändert.
```

```
// Es wird zwar beim "Auswerten" beachtet, aber nicht neu
// hinzugefügt. Dafür dient der Ereignis-Handle OnCombo3
void BasicFrame::ComboText( wxCommandEvent &event ) {
   wxComboBox *combo = (wxComboBox*) event.GetEventObject();
   wxString s = combo->GetValue();
   combo->SetValue(s);
}
```

wxCheckBox

Eine Checkbox ist ein Element, das normalerweise zwei Zustände besitzen kann: an oder aus. Gewöhnlich wird dieser Zustand (abhängig von der Plattform) durch ein Häkchen oder ein Kreuz und das Label rechts daneben angezeigt. Optional lässt sich noch ein dritter Zustand hinzufügen, indem man die Checkbox in einen unbestimmten Zustand (ausgegraut) setzt. Diese Option ist sinnvoll, wenn man ein Element aus der Checkbox nicht deaktivieren können soll, weil es dringend benötigt wird (wie dies zum Beispiel bei der Installation einer Software der Fall sein kann).

Der Konstruktor der Klasse `wxCheckBox` sieht folgendermaßen aus:

```
wxCheckBox( wxWindow* parent,
            wxWindowID id,
            const wxString& label,
            const wxPoint& pos = wxDefaultPosition,
            const wxSize& size = wxDefaultSize,
            long style = 0,
            const wxValidator& val,
            const wxString& name = "checkBox" );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster von `wxCheckBox` (sollte niemals `NULL` sein)
- `id` – die Identifizierung
- `label` – der Text, der rechts neben der Checkbox steht
- `pos` – die Position der Checkbox; bei (–1, –1) wird wieder die Default-Position verwendet.
- `size` – die Größe der Checkbox; bei (–1, –1) wird wieder die Default-Größe verwendet.
- `style` – der Checkbox-Stil (siehe Tabelle 10.31)
- `validator` – der Validator
- `name` – der Name für die Checkbox (wird nur unter Motif benötigt)

wxCheckBox Style	Beschreibung
wxCHK_2STATE	Erstellt eine Checkbox mit zwei möglichen Zuständen (Standardeinstellung).
wxCHK_3STATE	Erstellt eine Checkbox mit drei möglichen Zuständen.
wxCHK_ALLOW_3RD_STATE_FOR_USER	Dieses Flag bewirkt, dass auch der Anwender den dritten Zustand der Checkbox per Mausklick aktivieren bzw. deaktivieren kann.
wxALIGN_RIGHT	Soll die Checkbox auf der rechten Seite des Text-Labels stehen, können Sie dieses Flag verwenden.

Tabelle 10.31 Stile für »wxCheckBox«

In wxCheckBox finden Sie auch einige Methoden, die nur für Checkboxen von Bedeutung sind:

Methode	Beschreibung
`bool GetValue() const;`	Gibt den Zustand der Checkbox zurück. true wird zurückgegeben, wenn die Box aktiviert ist, und false, wenn nicht.
`wxCheckBoxState Get3StateValue() const;`	Gibt den Zustand einer Checkbox zurück, die alle drei Zustände haben kann. wxCHK_UNCHECKED wird zurückgegeben, wenn die Checkbox nicht aktiviert ist. wxCHK_CHECKED wird bei aktiviertem Zustand zurückgegeben, und wxCHK_UNDETERMINED wird zurückgegeben, wenn die Checkbox in einem unbestimmten Zustand ist (ausgegraut).
`bool Is3rdStateAllowedForUser() const;`	Gibt true zurück, wenn es dem Anwender erlaubt ist, den unbestimmten Zustand einer Checkbox zu aktivieren, ansonsten wird false zurückgegeben.
`bool Is3State() const;`	Gibt true zurück, wenn die Checkbox drei mögliche Zustände enthält, und false, wenn es sich um eine Checkbox mit zwei Zuständen handelt.
`bool IsChecked() const;`	Wie GetValue(), nur besser verständlich und lesbar.
`void SetValue(bool state);`	Übergibt der Checkbox beim Programmstart einen bestimmten Zustand. Gibt bei state true an, wenn die Checkbox beim Programmstart aktiviert ist, und false, wenn nicht. Diese Methode löst allerdings kein Ereignis aus.

Tabelle 10.32 Methoden von »wxCheckBox«

Methode	Beschreibung
`void Set3StateValue( const wxCheckBoxState state);`	Setzt die Checkbox, die drei Zustände annehmen kann, in einen bestimmten Zustand. Möglich sind hierbei `wxCHK_UNCHECKED`, wenn die Checkbox nicht aktiv sein soll, `wxCHK_CHECKED`, wenn sie aktiv sein soll, und `wxCHK_UNDETERMINED`, wenn die Checkbox in einem unbestimmten Zustand sein soll (ausgegraut). Auch hierbei löst diese Methode mit `wxCHK_CHECKED` kein Ereignis aus.

Tabelle 10.32 Methoden von »wxCheckBox« (Forts.)

`wxCheckBox` generiert ein Ereignis der Klasse `wxCommandEvent`. Auf das Ereignis können Sie mit einem Handle folgendermaßen reagieren:

Handle einrichten	Beschreibung
`EVT_CHECKBOX(id, func)`	Behandelt ein `wxEVT_COMMAND_CHECKBOX_CLICKED`-Ereignis, das generiert wird, wenn der Anwender die Checkbox aktiviert oder deaktiviert.

Tabelle 10.33 »wxCheckBox«-Ereignisse behandeln

In *base.cpp* erzeugen Sie im Konstruktor `BasicFrame` wie folgt ein Panel mit `wxCheckBox`:

```
wxPanel* window5 = CreatePanelWithCheckBox(choicebook);
```

Die globale Funktion `CreatePanelWithCheckBox`, in der `wxCheckBox` erzeugt wird, sieht wie folgt aus:

```
wxPanel *CreatePanelWithCheckBox(wxChoicebook *parent) {
   wxPanel *panel = new wxPanel(parent, wxID_ANY);
   // Ein Text-Label
   (void) new wxStaticText( panel, wxID_STATIC,
      wxT("Demonstriert Check-Boxen der Klasse wxCheckBox"),
      wxPoint(10, -1), wxDefaultSize, wxALIGN_LEFT );
   // Ein Rahmen um die Check-Box
   new wxStaticBox(
      panel, wxID_STATIC, wxT("Check-Box"),
      wxPoint(10,25) , wxSize(150,100));
   // Check-Boxen erzeugen
   wxCheckBox* m_checkbox1 = new wxCheckBox(
         panel, ID_CHECK_BUTTON1, wxT("Birne"),
         wxPoint(20,40), wxSize(-1, -1), wxCHK_3STATE  );
   wxCheckBox* m_checkbox2 = new wxCheckBox(
         panel, ID_CHECK_BUTTON2, wxT("Orange"),
         wxPoint(90,40), wxSize(-1, -1) );
```

```
   (void) new wxCheckBox(
         panel, ID_CHECK_BUTTON3, wxT("Apfel"),
         wxPoint(20,100), wxSize(-1, -1) );
   (void) new wxCheckBox(
         panel, ID_CHECK_BUTTON4, wxT("Kirsche"),
         wxPoint(90,100), wxSize(-1, -1) );
   // Zustand der Check-Boxen verändern
   m_checkbox1->Set3StateValue(wxCHK_UNDETERMINED);
   m_checkbox1->SetClientData( (wxChar*)
         wxT(" (Birne kann nicht deaktiviert werden) "));
   m_checkbox2->SetValue(true);
   return panel;
}
```

Dieses Panel fügen wir wieder zum *Choicebook* hinzu

```
choicebook->AddPage(window5, wxT("wxCheckBox"), false, 4);
```

und erhalten folgendes Bild bei der Ausführung:

Abbildung 10.26 »wxComboBox« bei der Ausführung

Hierzu richten wir wie folgt einen Ereignis-Handle ein:

```
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   ...
   EVT_CHECKBOX(ID_CHECK_BUTTON1, BasicFrame::OnCheck)
   EVT_CHECKBOX(ID_CHECK_BUTTON2, BasicFrame::OnCheck)
   EVT_CHECKBOX(ID_CHECK_BUTTON3, BasicFrame::OnCheck)
   EVT_CHECKBOX(ID_CHECK_BUTTON4, BasicFrame::OnCheck)
   ...
END_EVENT_TABLE()
```

Und hier noch der Code des Ereignis-Handles:

```
// Checkboxen auswerten
void BasicFrame::OnCheck( wxCommandEvent &event ) {
   wxCheckBox *combo1 = (wxCheckBox*)
      FindWindowById(ID_CHECK_BUTTON1);
   wxCheckBox *combo2 = (wxCheckBox*)
      FindWindowById(ID_CHECK_BUTTON2);
   wxCheckBox *combo3 = (wxCheckBox*)
      FindWindowById(ID_CHECK_BUTTON3);
   wxCheckBox *combo4 = (wxCheckBox*)
      FindWindowById(ID_CHECK_BUTTON4);

   wxString s(wxT("Ihre Auswahl: "));
   // "Birne" lässt sich nicht deaktivieren
   if( !(combo1->IsChecked()) ) {
      combo1->Set3StateValue(wxCHK_UNDETERMINED);
      s.Append( (wxChar*)combo1->GetClientData());
   }
   if( combo1->IsChecked())
      s.Append(wxT("Birne "));
   if( combo2->IsChecked())
      s.Append(wxT("Orange "));
   if( combo3->IsChecked())
      s.Append(wxT("Apfel "));
   if( combo4->IsChecked())
      s.Append(wxT("Kirsche "));
   SetStatusText( s );
}
```

»wxListBox« und »wxCheckListBox«

`wxListBox` ist eine einfache Liste von Strings, in der eine oder mehrere Auswahlen getroffen werden können. Die Liste der Strings wird dabei in einer gegebenenfalls scrollenden Box angezeigt.

`wxCheckListBox` wurde abgeleitet von `wxListBox` und besitzt dieselbe Funktionalität wie die Basisklasse. Nur wurde hierbei noch die Funktionalität der Checkbox hinzugefügt.

Zunächst die Syntax des Konstruktors von `wxListBox` (die es wieder in zwei Versionen gibt):

```
wxListBox( wxWindow* parent,
           wxWindowID id,
           const wxPoint& pos = wxDefaultPosition,
           const wxSize& size = wxDefaultSize,
           int n,
           const wxString choices[] = NULL,
```

```
            long style = 0,
            const wxValidator& validator=wxDefaultValidator,
            const wxString& name = "listBox" );

wxListBox( wxWindow* parent,
            wxWindowID id,
            const wxPoint& pos,
            const wxSize& size,
            const wxArrayString& choices,
            long style = 0,
            const wxValidator& validator=wxDefaultValidator,
            const wxString& name = "listBox" );
```

Die einzelnen Parameter von `wxListBox` haben folgende Bedeutung:

- `parent` – das Eltern-Fenster von `wxListBox` (sollte niemals `NULL` sein)
- `id` – die Identifizierung
- `pos` – die Position; bei (–1, –1) wird wieder die Default-Position verwendet.
- `size` – die Größe; bei (–1, –1) wird wieder die Default-Größe verwendet.
- Liste der Strings (Version 1):
  - `n` – Anzahl der Strings in der Liste
  - `choices` – ein Array mit Strings (`wxString`), die angezeigt werden sollen
- Liste der Strings (Version 2):
  - `choices` – ein Array mit Strings (`wxStringArray`), die angezeigt werden sollen
- `style` – der Stil von `wxListBox` (siehe Tabelle 10.34)
- `validator` – der Validator
- `name` – der Name für `wxChoice`

Die Syntax von `wxCheckListBox` können wir uns sparen, weil diese exakt der von `wxListBox` entspricht (mit denselben Stilen, und auch hier sind zwei Versionen vorhanden).

Stil	Beschreibung
`wxLB_SINGLE`	Es kann nur ein Element auf einmal ausgewählt werden.
`wxLB_MULTIPLE`	Es können mehrere Elemente auf einmal ausgewählt werden.
`wxLB_EXTENDED`	Mehrere Elemente können nur in Verbindung mit gedrückter ⇧-Taste ausgewählt werden.
`wxLB_HSCROLL`	Sollte der Inhalt zu breit werden, wird eine horizontale Scrollbar eingefügt.

Tabelle 10.34 Die Stile von »wxListBox« und »wxCheckListBox«

Stil	Beschreibung
wxLB_ALWAYS_SB	Die vertikale Scrollbar wird immer angezeigt.
wxLB_NEEDED_SB	Es wird nur eine vertikale Scrollbar erzeugt, wenn diese gebraucht wird.
wxLB_SORT	Die Elemente in der Liste werden alphabetisch sortiert und angezeigt.

Tabelle 10.34 Die Stile von »wxListBox« und »wxCheckListBox« (Forts.)

Beide Elemente haben einige Methoden gemeinsam. Diese Methoden werden in Tabelle 10.35 aufgelistet. Zusätzlich zu den Methoden von `wxListBox` besitzt `wxCheckListBox` noch einige Methoden, die in Tabelle 10.36 aufgelistet sind.

Methode	Beschreibung
`void Deselect(int n);`	Wählt das `n`-te Element (angefangen bei `0`) der Liste ab.
`int GetSelections(` `  wxArrayInt& selections) const;`	Damit können Sie über ein Array von `int` ermitteln, welche Elemente ausgewählt sind. Anschließend müssen Sie dieses Array auswerten. Im Array befinden sich dann der oder die entsprechende(n) Index(e) des oder der ausgewählten Elemente. Der Rückgabewert ist die Anzahl der selektierten Elemente. Natürlich hat dies nur bei Listen einen Sinn, bei denen mehrere Elemente ausgewählt werden können.

`void InsertItems(`
`  int nItems,`
`  const wxString *items,`
`  int pos );`
`void InsertItems(`
`  const wxArrayString& nItems,`
`  int pos );`
	Fügt eine bestimmte Anzahl von Strings für die angegebene Position `pos` ein.

`bool IsSelected(int n) const;`	Überprüft, ob das Element `n` in der Liste selektiert ist. Wenn ja, wird `true`, ansonsten `false` zurückgegeben.

`void Set(`
`  int n,`
`  const wxString* choices,`
`  void **clientData = NULL );`
 	
`void Set(`
`  const wxArrayString& choices,`
`  void **clientData = NULL );`
	Löscht die Listbox und fügt neue Strings hinzu. Hier können wiederum Client-Daten (siehe auch `wxControlWithItems`) hinzugefügt werden.

Tabelle 10.35 Methoden von »wxListBox« und »wxCheckListBox«

Methode	Beschreibung

`void SetFirstItem(int n);`
 	
`void SetFirstItem(`
`  const wxString& string );`
	Setzt ein bestimmtes Element als erstes sichtbares Element in der Liste.

Tabelle 10.35 Methoden von »wxListBox« und »wxCheckListBox« (Forts.)

Methode	Beschreibung
`void Check( int item,` `  bool check = true );`	Aktiviert das Element `item`, falls für `check` `true` (Standard) angegeben wird. Wird bei `check` `false` angegeben, wird das Element deaktiviert (Häkchen entfernt). Hierbei wird allerdings kein Ereignis ausgelöst.
`bool IsChecked(int item) const;`	Überprüft, ob das Element mit `item` aktiviert ist. Wenn `true` zurückgegeben wird, trifft dies zu, ansonsten wird `false` zurückgegeben.

Tabelle 10.36 Weitere Methoden von »wxCheckListBox«

`wxListBox` und `wxCheckListBox` generieren ein Ereignis der Klasse `wxCommandEvent`. Auf das Ereignis können Sie mit einem Handle folgendermaßen reagieren:

Handle einrichten	Beschreibung
`EVT_LISTBOX(id, func)`	Behandelt ein `wxEVT_COMMAND_LISTBOX_SELECTED`-Ereignis, das generiert wird, wenn der Anwender ein `wxListBox`-Element auswählt.
`EVT_LISTBOX_DCLICK(id, func)`	Behandelt ein `wxEVT_COMMAND_LISTBOX_DOUBLECLICKED`-Ereignis, das generiert wird, wenn der Anwender ein `wxListBox`-Element mit der Maus doppelt anklickt.
`EVT_CHECKLISTBOX (id, func)`	Behandelt ein `wxEVT_COMMAND_CHECKLISTBOX_TOGGLED`-Ereignis, das generiert wird, wenn der Anwender ein `wxCheckListBox`-Element aktiviert oder deaktiviert.

Tabelle 10.37 »wxCheckBox«-Ereignisse behandeln

Im Beispiel *base.cpp* verwenden wir im Konstruktor `BasicFrame` gleich beide Elemente in einem Panel. Hierzu rufen wir folgende Funktion auf:

```
wxPanel* window6 = CreatePanelWithListBox(choicebook);
```

Die globale Funktion `CreatePanelWithListBox`, in der `wxListBox` und `wxCheckListBox` erzeugt werden, sieht folgendermaßen aus:

```
wxPanel *CreatePanelWithListBox(wxChoicebook *parent) {
   wxPanel *panel = new wxPanel(parent, wxID_ANY);
```

```
    wxArrayString og;
    og.Add(wxT("Birne"));
    og.Add(wxT("Orange"));
    og.Add(wxT("Apfel"));
    og.Add(wxT("Kirsche"));
    og.Add(wxT("Tomate"));
    og.Add(wxT("Zwiebel"));
    og.Add(wxT("Lauch"));
    og.Add(wxT("Kartoffel"));
    // Ein Text-Label zum Panel
    (void) new wxStaticText( panel, wxID_STATIC,
         wxT("Demonstriert List-Boxen der Klasse wxListBox"),
         wxPoint(10, -1), wxDefaultSize, wxALIGN_LEFT );
    // Eine Listbox erzeugen
    (void) new wxListBox(
          panel, ID_LISTBOX,
          wxPoint(10,20), wxSize(150,80),
          og, wxLB_MULTIPLE );
    // Eine Check-List-Box erzeugen
    wxCheckListBox * checklistBox = new wxCheckListBox(
          panel, ID_CHECK_LISTBOX,
          wxPoint(190, 20), wxSize( 150, 80),
          og, wxLB_MULTIPLE );
    // Zustand verändern
    checklistBox->Check( 2 );
    return panel;
}
```

Dieses Panel fügen wir jetzt noch mit

```
choicebook->AddPage(window6, wxT("wxListBox"), false, 5 );
```

zum *Choicebook* hinzu und erhalten bei der Ausführung folgendes Bild:

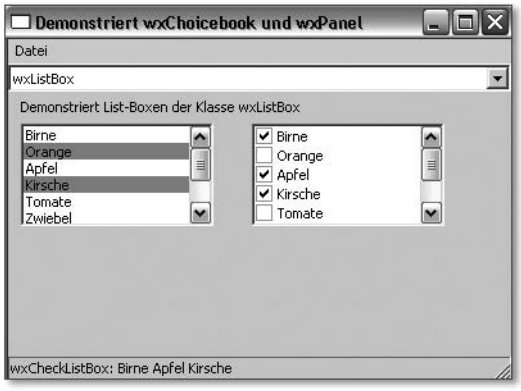

Abbildung 10.27 »wxListBox« und »wxCheckListBox« bei der Ausführung

Jetzt noch die Tabelle, in der der Ereignis-Handle eingerichtet wird,

```
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   ...
   EVT_LISTBOX(ID_LISTBOX, BasicFrame::OnListBox)
   EVT_CHECKLISTBOX(ID_CHECK_LISTBOX,
      BasicFrame::OnCheckListBox)
   ...
END_EVENT_TABLE()
```

und die Definitionen der Ereignis-Handles:

```
// Listboxen auswerten
void BasicFrame::OnListBox( wxCommandEvent &event ) {
   wxListBox *list = (wxListBox*)FindWindowById(ID_LISTBOX);
    wxString og[] = {
       wxT("Birne"), wxT("Orange"), wxT("Apfel"),
       wxT("Kirsche"), wxT("Tomate"), wxT("Zwiebel"),
       wxT("Lauch"), wxT("Kartoffel")
    };
   wxArrayInt select;
   wxString s(wxT("wxListBox: "));
   int n = list->GetSelections(select);
   for(int i = 0; i<n; i++) {
      s.Append( og[ select[i] ] );
      s.Append(wxT(" "));
   }
   SetStatusText( s );
}

// Check-Listboxen auswerten
void BasicFrame::OnCheckListBox( wxCommandEvent &event ) {
   wxCheckListBox *checklist = (wxCheckListBox*)
      FindWindowById(ID_CHECK_LISTBOX);
    wxString og[] = {
       wxT("Birne"), wxT("Orange"), wxT("Apfel"),
       wxT("Kirsche"), wxT("Tomate"), wxT("Zwiebel"),
       wxT("Lauch"), wxT("Kartoffel")
    };
   wxString s(wxT("wxCheckListBox: "));
   for( int i=0; i<8; i++) {
      if( checklist->IsChecked(i) ) {
         s.Append(og[i]);
         s.Append(wxT(" "));
       }
   }
```

```
   SetStatusText( s );
}
```

»wxRadioBox« und »wxRadioButton«

Eine Radio-Box ist der Checkbox recht ähnlich. Nur wird hierbei aus einer Reihe von Elementen eines ausgewählt. Anstelle eines Häkchens wird ein einfacher Punkt dargestellt. Das Text-Label befindet sich gewöhnlich auf der rechten Seite des Radio-Buttons.

In `wxWidgets` stehen Ihnen zwei Möglichkeiten zur Verfügung, solche Radio-Buttons einzusetzen – zum einen mit der Klasse `wxRadioBox`, zum anderen mit `wxRadioButton`. Üblicherweise greift man auf `wxRadioBox` zurück. Hiermit lässt sich nebenbei auch ganz komfortabel die Anzahl der Reihen oder Spalten festlegen, in denen die Buttons angeordnet werden sollen.

`wxRadioButton` hingegen wird verwendet, wenn das Layout der Anwendung etwas komplexer ist und nicht erlaubt, die Klasse `wxRadioBox` einzusetzen. Mit `wxRadioButton` können Sie jeden einzelnen Button erzeugen und verwalten. Mit `wxRadioBox` dagegen verwalten Sie die ganze Gruppe von Radio-Buttons.

Zunächst die Syntax für `wxRadioBox` (in zweifacher Ausführung vorhanden):

```
wxRadioBox( wxWindow* parent,
            wxWindowID id,
            const wxString& label,
            const wxPoint& pos = wxDefaultPosition,
            const wxSize& size = wxDefaultSize,
            int n = 0,
            const wxString choices[] = NULL,
            int majorDimension = 0,
            long style = wxRA_SPECIFY_COLS,
            const wxValidator& validator=wxDefaultValidator,
            const wxString& name = "radioBox");

wxRadioBox( wxWindow* parent,
            wxWindowID id,
            const wxString& label,
            const wxPoint& pos,
            const wxSize& size,
            const wxArrayString& choices,
            int majorDimension = 0,
            long style = wxRA_SPECIFY_COLS,
            const wxValidator& validator=wxDefaultValidator,
            const wxString& name = "radioBox" );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster von `wxRadioBox` (sollte niemals `NULL` sein)
- `id` – die Identifizierung
- `label` – der Text für die statische Box, die rund um die Radio-Box gezogen wird
- `pos` – die Position; bei (–1, –1) wird wieder die Default-Position verwendet.
- `size` – die Größe; bei (–1, –1) wird wieder die Default-Größe verwendet.
- **Liste der Strings (Version 1):**
  - `n` – Anzahl der Strings in der Liste
  - `choices` – ein Array mit Strings (`wxString`), die angezeigt werden sollen
- **Liste der Strings (Version 2):**
  - `choices` – ein Array mit Strings (`wxStringArray`), die angezeigt werden sollen
- `majorDimension` – gibt die maximale Anzahl von Zeilen (wenn `style=wxRA_SPECIFY_ROWS`) oder Spalten (wenn `style= wxRA_SPECIFY_COLS`) für die zweidimensionale Radio-Box an.
- `style` – der Stil von `wxRadioBox` (siehe Tabelle 10.38)
- `validator` – der Validator
- `name` – der Name für `wxRadioBox`

wxRadioBox style	Beschreibung
`wxRA_SPECIFY_ROWS`	Der Wert des Parameters `majorDimension` legt die maximale Anzahl von Zeilen der Radio-Box fest.
`wxRA_SPECIFIY_COLS`	Der Wert des Parameters `majorDimension` legt die maximale Anzahl von Spalten der Radio-Box fest.

Tabelle 10.38 Stile von »wxRadioBox«

Die Klasse `wxRadioBox` bietet auch einige bedeutende Methoden, die in Tabelle 10.39 näher beschrieben werden.

Methode	Beschreibung
`virtual bool Enable( int n, bool enable = true );`	Mit dieser Methode können Sie einzelne Einträge der Radio-Box aktivieren (`enable=true`) oder deaktivieren (`enable=false`). Mit `n` geben Sie den entsprechenden Radio-Button an.

Tabelle 10.39 Methoden von »wxRadioBox«

Methode	Beschreibung
`virtual bool Enable( bool enable = true );`	Mit dieser Methode können Sie alle Einträge der Radio-Box aktivieren (`enable=true`) oder deaktivieren (`enable=false`). Die Radio-Box kann somit im deaktivierten Zustand nicht benutzt werden.
`int FindString( const wxString& string) const;`	Sucht nach einem String in der Radio-Box und gibt die entsprechende Position zurück oder, falls nicht gefunden, `–1`.
`Int GetColumnCount() const;`	Gibt die Anzahl der Spalten der Radio-Box zurück.
`Int GetCount() const;`	Gibt die Anzahl der Einträge der Radio-Box zurück.
`wxString GetLabel() const;`	Gibt das Text-Label der Radio-Box zurück.
`int GetRowCount() const;`	Gibt die Anzahl der Zeilen der Radio-Box zurück.
`int GetSelection() const;`	Gibt die Position des ausgewählten Radio-Buttons zurück.
`wxString GetStringSelection() const;`	Gibt den String des ausgewählten Radio-Buttons zurück.
`wxString GetString( int n ) const;`	Gibt den String des Radio-Buttons an Position `n` zurück.
`void SetLabel( const wxString& label );`	Setzt das Text-Label um die Radio-Box.
`void SetSelection(int n);`	Setzt den Radio-Button an der Position `n` als angewählt. Hierbei wird allerdings kein Ereignis ausgelöst.
`void SetStringSelection( const wxString& string );`	Setzt den Radio-Button mit dem String `string` als angewählt. Hierbei wird allerding kein Ereignis ausgelöst.
`virtual bool Show( int item, const bool show = true );`	Damit lassen sich einzelne Einträge der Radio-Box anzeigen (`true`) oder verstecken (`false`).
`virtual bool Show( const bool show = true );`	Zeigt oder versteckt die Radio-Box. Mit `true` wird die Radio-Box angezeigt und mit `false` versteckt.

Tabelle 10.39 Methoden von »wxRadioBox« (Forts.)

`wxRadioBox` generiert ein Ereignis der Klasse `wxCommandEvent`. Auf das Ereignis können Sie mit einem Handle folgendermaßen reagieren:

Handle einrichten	Beschreibung
`EVT_RADIOBOX(id, func)`	Behandelt ein `wxEVT_COMMAND_RADIOBOX_SELECTED`-Ereignis, das generiert wird, wenn der Anwender ein `wxRadioBox`-Element auswählt.

Tabelle 10.40 »wxRadioBox«-Ereignisse behandeln

Der Konstruktor für wxRadioButton sieht so aus:

```
wxRadioButton(
            wxWindow* parent,
            wxWindowID id,
            const wxString& label,
            const wxPoint& pos = wxDefaultPosition,
            const wxSize& size = wxDefaultSize,
            long style = 0,
            const wxValidator& validator=wxDefaultValidator,
            const wxString& name = "radioButton"  );
```

Die einzelnen Parameter haben folgende Bedeutung:

- parent – das Eltern-Fenster von wxRadioButton (sollte niemals NULL sein)
- id – die Identifizierung
- label – der Text für den Radio-Button
- pos – die Position; bei (–1, –1) wird wieder die Default-Position verwendet.
- size – die Größe; bei (–1, –1) wird wieder die Default-Größe verwendet.
- style – der Stil von wxRadioButton (siehe Tabelle 10.41)
- validator – der Validator
- name – der Name für wxRadioButton

wxRadioButton style	Beschreibung
wxRB_GROUP	Markiert den Anfang einer neuen Gruppe von Radio-Buttons.
wxRB_USE_CHECKBOX	Verwendet einen Check-Button anstelle eines Radio-Buttons (nur für Palm OS).

Tabelle 10.41 Die Stile von »wxRadioButton«

Die Klasse wxRadioButton hat nur zwei Methoden, die in Tabelle 10.42 aufgelistet sind:

Methode	Beschreibung
`bool GetValue() const;`	Gibt true zurück, wenn der Radio-Button ausgewählt wurde, ansonsten wird false zurückgegeben.
`void SetValue( const bool value );`	Setzt den Radio-Button auf einen ausgewählten (true) oder nicht ausgewählten (false) Zustand. Hierbei wird allerdings kein Ereignis ausgelöst.

Tabelle 10.42 Methoden von »wxRadioButton«

wxRadioButton generiert ebenfalls ein Ereignis der Klasse wxCommandEvent. Auf das Ereignis können Sie mit einem Handle folgendermaßen reagieren:

Handle einrichten	Beschreibung
`EVT_RADIOBUTTON(id, func)`	Behandelt ein `wxEVT_COMMAND_RADIOBUTTON_SELECTED`-Ereignis, das generiert wird, wenn der Anwender ein `wxRadioButton`-Element auswählt.

Tabelle 10.43 »wxRadioButton«-Ereignisse behandeln

Jetzt können wir im Beispiel *base.cpp* wieder in `BasicFrame` die beiden Elemente zu einem Panel hinzufügen:

```
wxPanel* window7 = CreatePanelWithRadioBox(choicebook);
```

In der globalen Funktion `CreatePanelWithRadioBox` erzeugen wir gleich beide hier vorgestellten Elemente `wxRadioBox` und `wxRadioButton`. Hier die Funktion:

```
wxPanel *CreatePanelWithRadioBox(wxChoicebook *parent) {
   wxPanel *panel = new wxPanel(parent, wxID_ANY);
   wxArrayString og;
   og.Add(wxT("Birne"));
   og.Add(wxT("Orange"));
   og.Add(wxT("Apfel"));
   og.Add(wxT("Kirsche"));
   og.Add(wxT("Tomate"));
   og.Add(wxT("Zwiebel"));
   og.Add(wxT("Lauch"));
   og.Add(wxT("Kartoffel"));
   // Ein Text-Label zum Panel
   (void) new wxStaticText(panel, wxID_STATIC,
         wxT("Demonstriert Radio-Boxen und -Buttons")
         wxT(" von wxRadioBox und wxRadioButton"),
         wxPoint(10, -1), wxDefaultSize, wxALIGN_LEFT);
   // Eine Radio-Box erzeugen
   wxRadioBox* radioBox = new wxRadioBox(
         panel, ID_RADIOBOX, wxT("Radio-Box"),
         wxPoint(10, 20), wxDefaultSize,
         og, 2, wxRA_SPECIFY_COLS);
   // "Apfel" vorauswählen ...
   radioBox->SetStringSelection(wxT("Apfel"));
   // "Tomate" deaktivieren
   radioBox->Enable(4, false);
   // Einen Rahmen erzeugen
   (void) new wxStaticBox(
         panel, wxID_STATIC, wxT("Radio-Buttons"),
         wxPoint(170,20), wxSize(150,90));
   // Radio-Buttons erzeugen
   wxRadioButton* radioButton1 = new wxRadioButton (
```

```
            panel, ID_RADIOBUTTON1, wxT("Vorspeise"),
            wxPoint(180, 40), wxDefaultSize, wxRB_GROUP);
      // Aktivieren ...
      radioButton1->SetValue(true);
      (void) new wxRadioButton (
            panel, ID_RADIOBUTTON2, wxT("Hauptspeise"),
            wxPoint(180,60) );
      (void) new wxRadioButton (
            panel, ID_RADIOBUTTON3, wxT("Nachspeise"),
            wxPoint(180,80));
      return panel;
}
```

Jetzt noch zum *Choicebook* hinzufügen

```
choicebook->AddPage(window7, wxT("wxRadioBox"), false, 6 );
```

und schon haben wir die nächste Abbildung (siehe Abbildung 10.28).

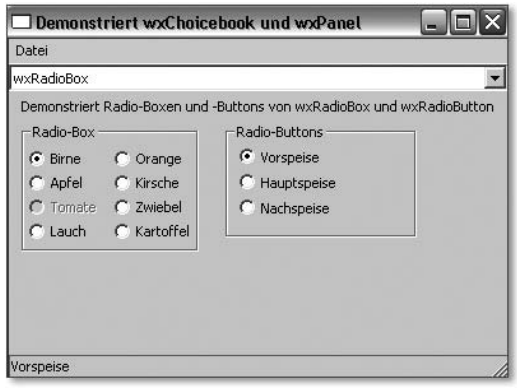

Abbildung 10.28 »wxRadioBox« und »wxRadioButton« bei der Ausführung

Jetzt fehlen nur noch die Ereignis-Handles. Zunächst wieder die Tabelle:

```
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   ...
   EVT_RADIOBOX(ID_RADIOBOX, BasicFrame::OnRadioBox)
   EVT_RADIOBUTTON(ID_RADIOBUTTON1,
      BasicFrame::OnRadioButton)
   EVT_RADIOBUTTON(ID_RADIOBUTTON2,
      BasicFrame::OnRadioButton)
   EVT_RADIOBUTTON(ID_RADIOBUTTON3,
      BasicFrame::OnRadioButton)
   ...
END_EVENT_TABLE()
```

Und jetzt noch die Definition der einzelnen Handles:

```
// Radio-Boxen auswerten
void BasicFrame::OnRadioBox( wxCommandEvent &event ) {
   wxRadioBox *radio = (wxRadioBox*)
      FindWindowById(ID_RADIOBOX);
   SetStatusText( radio->GetStringSelection() );
}

// Radio-Button auswerten
void BasicFrame::OnRadioButton( wxCommandEvent &event ) {
   wxRadioButton *radio1 = (wxRadioButton*)
      FindWindowById(ID_RADIOBUTTON1);
   wxRadioButton *radio2 = (wxRadioButton*)
      FindWindowById(ID_RADIOBUTTON2);
   wxRadioButton *radio3 = (wxRadioButton*)
      FindWindowById(ID_RADIOBUTTON3);
   if(radio1->GetValue())
      SetStatusText( wxT("Vorspeise") );
   if(radio2->GetValue())
      SetStatusText( wxT("Hauptspeise") );
   if(radio3->GetValue())
      SetStatusText( wxT("Nachspeise") );
}
```

»wxSpinCtrl« und »wxSlider«

`wxSpinCtrl` kombiniert die Klassen `wxTextCtrl` und `wxSpinButton`. `wxSpinButton` besitzt zwei kleine Buttons mit einem Pfeil nach oben und einem Pfeil nach unten (oder rechts und links). Wenn Sie auf den Button mit dem Pfeil nach oben klicken, wird der Wert, der in `wxTextCtrl` angezeigt wird, inkrementiert. Natürlich können Sie zusätzlich den Wert im Textfeld von Hand verändern. Da `wxSpinButton` nicht auf allen Plattformen implementiert ist, wird angeraten, immer `wxSpinCtrl` zu verwenden.

Ein *Slider* (`wxSlider`; dt.: Schieber) ist ein Element, bei dem Sie einen Hebel nach rechts oder links (bzw. oben oder unten) bewegen können, um so einen Wert zu verändern.

Beginnen wollen wir mit dem Konstruktor von `wxSpinCtrl`:

```
wxSpinCtrl( wxWindow* parent,
            wxWindowID id = -1,
            const wxString& value = wxEmptyString,
            const wxPoint& pos = wxDefaultPosition,
            const wxSize& size = wxDefaultSize,
            long style = wxSP_ARROW_KEYS,
```

```
    int min = 0,
    int max = 100,
    int initial = 0,
    const wxString& name = _T("wxSpinCtrl") );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster von `wxSpinCtrl` (sollte niemals `NULL` sein)
- `id` – die Identifizierung
- `value` – der Vorgabewert, der im Textfeld angegeben werden soll. Hier muss nicht zwangsläufig ein Zahlenwert stehen.
- `pos` – die Position; bei (–1, –1) wird wieder die Default-Position verwendet.
- `size` – die Größe; bei (–1, –1) wird wieder die Default-Größe verwendet.
- `style` – der Stil von `wxSpinCtrl` (siehe Tabelle 10.44)
- `min` – der kleinstmögliche Wert
- `max` – der größtmögliche Wert
- `initial` – der Wert, mit dem `wxSpinCtrl` beim Programmstart vorbelegt ist
- `name` – der Name für `wxSpinCtrl`

wxSpinCtrl style	Beschreibung
`wxSP_ARROW_KEYS`	Sie können die Pfeiltasten der Tastatur verwenden, um den Wert zu verändern.
`wxSP_WRAP`	Ist der maximale oder der minimale Wert erreicht, beginnt der Wert wieder von vorn. Ist der Bereich zum Beispiel 0 – 100, und es wurde der Wert 100 erreicht und wird der Wert dann um 1 inkrementiert, beginnt der Zähler wieder bei 0.

Tabelle 10.44 Die Stile von »wxSpinCtrl«

Die Methoden von `wxSpinCtrl` sind in Tabelle 10.45 beschrieben.

Methode	Beschreibung
`void SetValue( const wxString& text); void SetValue(int value);`	Setzt den Wert im Textfeld auf `text` oder `value`.
`int GetValue() const;`	Ermittelt den aktuellen Wert von `wxSpinCtrl`.
`void SetRange( int minVal, int maxVal );`	Setzt den minimalen und maximalen Wert von `wxSpinCtrl`.
`int GetMin() const;`	Ermittelt den minimal möglichen Wert von `wxSpinCtrl`.

Tabelle 10.45 Methoden von »wxSpinCtrl«

Methode	Beschreibung
`int GetMax() const;`	Ermittelt den maximal möglichen Wert von `wxSpinCtrl`.

Tabelle 10.45 Methoden von »wxSpinCtrl« (Forts.)

`wxSpinCtrl` generiert ebenfalls ein Ereignis der Klasse `wxCommandEvent`. Auf das Ereignis können Sie mit einem Handle folgendermaßen reagieren:

Handle einrichten	Beschreibung
`EVT_SPIN(id, func)`	Behandelt ein `wxEVT_SCROLL_THUMBTRACK`-Ereignis, das generiert wird, wenn der Anwender den nach oben oder den nach unten gerichteten Pfeil anklickt.
`EVT_SPIN_UP(id, func)`	Behandelt ein `wxEVT_SCROLL_LINEUP`-Ereignis, das generiert wird, wenn der Anwender auf den nach oben gerichteten Pfeil klickt.
`EVT_SPIN_DOWN(id, func)`	Behandelt ein `wxEVT_SCROLL_LINEDOWN`-Ereignis, das generiert wird, wenn der Anwender auf den nach unten gerichteten Pfeil klickt.
`EVT_SPINCTRL(id, func)`	Behandelt alle Ereignisse von `wxSpinCtrl`.

Tabelle 10.46 »wxSpinCtrl«-Ereignisse behandeln

Hinweis [«]

Sie können auch mit `EVT_TEXT` – wenn der Text erneuert wurde – einen Ereignis-Handle mit `wxCommandEvent` einrichten.

Weiter mit dem Konstruktor von `wxSlider`:

```
wxSlider( wxWindow* parent,
          wxWindowID id,
          int value,
          int minValue,
          int maxValue,
          const wxPoint& pos = wxDefaultPosition,
          const wxSize& size = wxDefaultSize,
          long style = wxSL_HORIZONTAL,
          const wxValidator& validator = wxDefaultValidator,
          const wxString& name = "slider" );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster von `wxSlider` (sollte niemals `NULL` sein)
- `id` – die Identifizierung

- `value` – der Wert, mit dem `wxSlider` bei Programmstart initialisiert wird
- `minValue` – die minimale Slider-Position
- `maxValue` – die maximale Slider-Position
- `pos` – die Position; bei (–1, –1) wird wieder die Default-Position verwendet.
- `size` – die Größe; bei (–1, –1) wird wieder die Default-Größe verwendet.
- `style` – der Stil von `wxSlider` (siehe Tabelle 10.47)
- `validator` – der Validator
- `name` – der Name für `wxSlider`

wxSlider style	Beschreibung
`wxSL_HORIZONTAL`	Zeigt einen horizontalen Slider.
`wxSL_VERTICAL`	Zeigt einen vertikalen Slider.
`wxSL_AUTOTICKS`	Zeigt die einzelnen Ticks (Striche) auf dem Slider an.
`wxSL_LABELS`	Zeigt die maximalen, minimalen und aktuellen Werte als Text-Label an.
`wxSL_LEFT`	Zeigt die Ticks (Striche) auf der linken Seite eines vertikalen Sliders an.
`wxSL_RIGHT`	Zeigt die Ticks (Striche) auf der rechten Seite eines vertikalen Sliders an.
`wxSL_TOP`	Zeigt die Ticks (Striche) oben auf einem horizontalen Slider an.

Tabelle 10.47 Die Stile von »wxSlider«

Jetzt zu den Methoden von `wxSlider`, die Sie in Tabelle 10.48 finden.

Methode	Beschreibung
`int GetLineSize() const;`	Gibt die Länge der Linie zurück.
`int GetMax() const;`	Gibt den maximal möglichen Wert des Sliders zurück.
`int GetMin() const;`	Gibt den minimal möglichen Wert des Sliders zurück.
`int GetPageSize() const;`	Gibt die Seitengröße des Sliders zurück.
`int GetValue() const;`	Gibt den aktuellen Wert des Sliders zurück. Dieser Wert ist die Anzahl der Ticks (Striche), die ein Slider bewegt wird, wenn der Anwender irgendwo im Slider klickt und den Wert erhöht oder reduziert (nicht am Schieber des Sliders).

Tabelle 10.48 Methoden von »wxSlider«

Methode	Beschreibung
`void SetPageSize(int pageSize);`	Setzt die Seitengröße des Sliders auf `pageSize`. Dieser Wert ist die Anzahl der Ticks (Striche), die ein Slider bewegt wird, wenn der Anwender irgendwo im Slider klickt und den Wert erhöht oder reduziert (nicht am Schieber des Sliders).
`void SetRange(int minValue, int maxValue);`	Setzt den minimal und den maximal möglichen Wert des Sliders.
`void SetValue(int value);`	Setzt die Position des Sliders auf `value`.

Tabelle 10.48 Methoden von »wxSlider« (Forts.)

Hinweis **[«]**

In der Dokumentation finden Sie noch einige weitere `Get`- und `Set`-Zugriffsmethoden, die zwar alle für »Windows 95 only« ausgegeben sind, aber auch auf anderen Systemen funktionieren.

`wxSlider` generiert ebenfalls ein Ereignis der Klasse `wxCommandEvent`. Auf das Ereignis können Sie mit einem Handle folgendermaßen reagieren:

Methode	Beschreibung
`EVT_SLIDER(id, func)`	Behandelt ein `wxEVT_COMMAND_SLIDER_UPDATED`-Ereignis, das generiert wird, wenn der Anwender den Schieber des Sliders verändert.

Tabelle 10.49 wxSpinCtrl-Ereignisse behandeln

Hinweis **[«]**

Sie können die Kontrolle der Ereignisse nochmals erheblich verfeinern, indem Sie `EVT_COMMAND_SCROLL_...` verwenden, die ein Ereignis der Klasse `wxScrollEvent` behandeln. Hierzu sei bei Bedarf die Lektüre der Dokumentation empfohlen.

Jetzt kommen wir wieder zur Praxis und fügen `wxSpinCtrl` und `wxSlider` zum Quellcode *base.cpp* hinzu. Zunächst erzeugen wir wieder ein Panel mit beiden Elementen:

```
wxPanel* window8 = CreatePanelWithSpinCtrl(choicebook);
```

Hier die komplette Funktion `CreatePanelWithSpinCtrl`:

```
// Erzeugt ein Panel mit einer Spin-Kontrolle
// und Slider und gibt dies zurück
wxPanel *CreatePanelWithSpinCtrl(wxChoicebook *parent) {
   wxPanel *panel = new wxPanel(parent, wxID_ANY);
   // Ein Text-Label zum Panel
```

```
    (void) new wxStaticText( panel, wxID_STATIC,
          wxT("Demonstriert die Klasse wxSpinCtrl"),
          wxPoint(10, -1), wxDefaultSize, wxALIGN_LEFT );
    // Ein wxSpinCtrl-Element erzeugen
    (void) new wxSpinCtrl(
          panel, ID_SPINCTRL, wxT("Los geht es!"),
          wxPoint(10, 20) , wxDefaultSize,
          wxSP_ARROW_KEYS|wxSP_WRAP, 0, 100, 5);
    // Ein Text-Label
    (void) new wxStaticText( panel, wxID_STATIC,
          wxT("Demonstriert die Klasse wxSlider"),
          wxPoint(10, 50), wxDefaultSize, wxALIGN_LEFT);
    // Ein wxSlider erzeugen
    (void) new wxSlider(
          panel, ID_SLIDER, 10, 0, 20,
          wxPoint(10, 70), wxSize(200, -1),

          wxSL_AUTOTICKS | wxSL_LABELS  );
    return panel;
}
```

Das Panel fügen wir jetzt nur noch zum *Choicebook* hinzu

```
choicebook->AddPage(window8, wxT("wxSpinCtrl"), false, 7 );
```

und erhalten bei der Ausführung das Bild aus Abbildung 10.29.

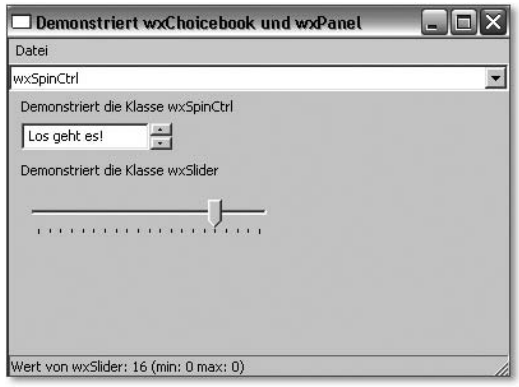

Abbildung 10.29 »wxSpinCtrl« und »wxSlider« bei der Ausführung

Als Nächstes fügen wir wieder die Ereignis-Handles der Tabelle hinzu:

```
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
    ...
```

```
   EVT_SPINCTRL(ID_SPINCTRL, BasicFrame::OnSpinCtrl)
   EVT_SLIDER(ID_SLIDER, BasicFrame::OnSlider )
   ...
END_EVENT_TABLE()
```

Jetzt noch die Definition der einzelnen Handles:

```
// Spin-Kontroll auswerten
void BasicFrame::OnSpinCtrl( wxSpinEvent &event ) {
   wxSpinCtrl *spin = (wxSpinCtrl*)
      FindWindowById(ID_SPINCTRL);
   wxString s = wxT("Wert von wxSpinCtrl: ");
   s += wxString::Format( wxT("%d (min: %d max: %d)"),
      spin->GetValue(), spin->GetMin(), spin->GetMax());
   SetStatusText( s );
}

// Slider auswerten
void BasicFrame::OnSlider( wxCommandEvent &event ) {
   wxSlider *slider = (wxSlider*) FindWindowById(ID_SLIDER);
   wxString s = wxT("Wert von wxSlider: ");
   s += wxString::Format( wxT("%d (min: %d max: %d)"),
      slider->GetValue(),slider->GetMin(),slider->GetMax());
   SetStatusText( s );
}
```

wxTextCtrl

`wxTextCtrl` wird verwendet, wenn Sie einen Text anzeigen und editieren wollen, sowohl einzeilig als auch mehrzeilig. Zum Formatieren (Farbe, Schriftart etc.) der Textattribute wird die Klasse `wxTextAttr` verwendet.

Hinweis **[«]**

Auf `wxTextCtrl` wird in diesem Abschnitt nur kurz eingegangen, da wir im Verlauf des Kapitels dieses Widget noch in einem umfangreichen Beispiel verwenden werden. Denoch sollen hier schon einmal die Methoden, Stile und das Einrichten des Ereignis-Handles beschrieben werden.

Zunächst der Konstruktor von `wxTextCtrl`:

```
wxTextCtrl( wxWindow* parent,
            wxWindowID id,
            const wxString& value = "",
            const wxPoint& pos = wxDefaultPosition,
            const wxSize& size = wxDefaultSize,
            long style = 0,
```

```
        const wxValidator& validator=wxDefaultValidator,
        const wxString& name = wxTextCtrlNameStr );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster von `wxTextCtrl` (sollte niemals `NULL` sein)
- `id` – die Identifizierung
- `value` – soll das Textfeld mit einem String vorbelegt werden, können Sie hier etwas angeben.
- `pos` – die Position; bei (–1, –1) wird wieder die Default-Position verwendet.
- `size` – die Größe; bei (–1, –1) wird wieder die Default-Größe verwendet.
- `style` – der Stil von `wxTextCtrl` (siehe Tabelle 10.50)
- `validator` – der Validator
- `name` – der Name für `wxTextCtrl`

wxTextCtrl style	Beschreibung
`wxTE_PROCESS_ENTER`	Damit können Sie das `wxEVT_COMMAND_TEXT_ENTER`-Ereignis behandeln, das ausgelöst wird, wenn im Textfeld ↵ gedrückt wurde.
`wxTE_PROCESS_TAB`	Wird die Tabulator-Taste gedrückt, wird ein paar Zeichen eingerückt (wie viel, ist abhängig vom System) – wie man dies von Text-Editoren kennt. Allerdings wird für ein Tabulator-Zeichen ein Zeichen verwendet. Gewöhnlich wird das Tabulator-Zeichen verwendet, um die einzelnen Kontrollelemente von `wxWidgets` anzuspringen.
`wxTE_MULTILINE`	Das Textfeld zum Editieren darf mehrere Zeilen haben.
`wxTE_PASSWORD`	Anstelle von Text werden Sternchen ausgegeben. Dies wird zum Beispiel bei der Eingabe von Passwörtern verwendet, damit sie für andere nicht kenntlich sind.
`wxTE_READONLY`	Der Text ist nur lesbar und kann nicht editiert werden.
`wxTE_RICH`	Verwendet das Rich-Text-Format (kurz RTF, nur MS Windows). Bei anderen Plattformen wird dieser Stil ignoriert.
`wxTE_RICH2`	Verwendet das Rich-Text-Format der Version 2.0 oder 3.0 unter MS Windows.
`wxTE_AUTO_URL`	Wenn Sie im Text eine URL angeben (beispielsweise »http://www.abc.de«), wird ein `wxTextUrlEvent`-Ereignis generiert. Gewöhnlich wird hierbei der Text unterstrichen und kann mit der Maus angeklickt werden. Unter MS Windows wird hierfür das Rich-Text- Format benötigt (nur für MS Windows und GTK+).

Tabelle 10.50 Die Stile von »wxTextCtrl«

wxTextCtrl style	Beschreibung
wxTE_NOHIDESEL	Wenn ein Text markiert wurde und das Fenster den Fokus verliert, wird diese Markierung nicht mehr angezeigt. Verwenden Sie diese Option, wird die Markierung immer angezeigt.
wxHSCROLL	Es wird eine horizontale Scrollbar erzeugt, so dass der Text niemals umbrochen wird.
wxTE_LEFT	Der Text im Textfeld ist links ausgerichtet (Standard).
wxTE_CENTRE	Der Text im Textfeld wird zentriert.
wxTE_RIGHT	Der Text im Textfeld ist rechts ausgerichtet.
wxTE_DONTWRAP	wie wxHSCROLL
wxTE_NO_VSCROLL	Entfernt die vertikale Scrollbar.

Tabelle 10.50 Die Stile von »wxTextCtrl« (Forts.)

Jetzt zu den Methoden, von denen die Klasse `wxTextCtrl` eine Menge anbietet. Einen Überblick dazu finden Sie in Tabelle 10.51.

Methode	Beschreibung
`void AppendText( const wxString& text );`	Fügt den Text `text` am Ende des Textfelds ein. Anschließend befindet sich der »Einfügepunkt« am Ende des Textfelds. Wenn dies nicht gewünscht ist, sollte man die Methoden `GetInsertionPoint` und `SetInsertionPoint` verwenden.
`virtual bool CanCopy();`	Gibt `true` zurück, wenn die Textauswahl in die Zwischenablage kopiert werden kann, ansonsten wird `false` zurückgegeben.
`virtual bool CanCut();`	Gibt `true` zurück, wenn die Textauswahl ausgeschnitten und in die Zwischenablage kopiert werden kann – ansonsten wird `false` zurückgegeben.
`virtual bool CanPaste();`	Gibt `true` zurück, wenn der Inhalt der Zwischenablage in das Textfeld eingefügt werden kann, ansonsten wird `false` zurückgegeben.
`virtual bool CanRedo();`	Gibt `true` zurück, wenn die zuvor durchgeführte Operation (bezüglich des Textfelds) nochmals ausgeführt werden kann, ansonsten wird `false` zurückgegeben.
`virtual bool CanUndo();`	Gibt `true` zurück, wenn der zuvor durchgeführte Vorgang (bezüglich des Textfelds) rückgängig gemacht werden kann, ansonsten wird `false` zurückgegeben.
`virtual void Clear();`	Löscht den Text im Textfeld. Hierbei wird außerdem das Ereignis `wxEVT_COMMAND_TEXT_UPDATED` generiert.

Tabelle 10.51 Die Methoden von »wxTextCtrl«

Methode	Beschreibung
`virtual void Copy();`	Kopiert den ausgewählten Text in die Zwischenablage.
`virtual void Cut();`	Schneidet den ausgewählten Text aus und kopiert diesen in die Zwischenablage.
`void DiscardEdits();`	Setzt das `modified`-Flag zurück. Ein editierter Text wird hiermit als bereits gespeichert markiert.
`const wxTextAttr& GetDefaultStyle() const;`	Gibt den aktuellen Textstil zurück, der für den neuen Text verwendet wird.
`virtual long GetInsertionPoint() const;`	Gibt die Position rechts neben der Einfügeposition (angefangen bei 0) zurück.
`virtual wxTextPos GetLastPosition() const;`	Gibt die letztmögliche Position des Textfelds zurück – entspricht somit der Anzahl der Zeichen im Textfeld.
`int GetLineLength( long lineNo ) const;`	Gibt die Länge der Zeile mit der Nummer `lineNo` ohne Newline-Zeichen zurück. Ist die Angabe zu `lineNo` falsch, wird `-1` zurückgegeben.
`wxString GetLineText( long lineNo) const;`	Gibt den kompletten String (Text) der Zeile `lineNo` ohne Newline-Zeichen zurück.
`int GetNumberOfLines() const;`	Gibt die Anzahl von Zeilen des Textfelds zurück. Beachten Sie, dass der Rückgabewert nie `0` ist. Selbst ein leeres Textfeld gibt eine Zeile zurück (die auch der Einfügepunkt ist).
`virtual wxString GetRange( long from, long to) const;`	Gibt den String (Text) zwischen der Position `from` und `to` zurück.
`virtual void GetSelection( long* from, long* to) const;`	Gibt die Positionen des aktuell selektierten Textbereichs an die Adressen `from` bis `to` zurück. Sind beide zurückgegebenen Werte gleich (`from==to`), wurde nichts selektiert.
`virtual wxString GetStringSelection();`	Gibt den ausgewählten Text zurück. Wurde kein Text ausgewählt, ist der String leer.
`bool GetStyle( long position, wxTextAttr& style );`	Gibt den Stil an der Position `position` des Textfelds zurück.
`wxString GetValue() const;`	Gibt den kompletten Inhalt des Textfelds zurück.
`bool IsEditable() const;`	Gibt `true` zurück, wenn der Text editierbar ist, ansonsten wird `false` zurückgegeben.
`bool IsMultiLine() const;`	Gibt `true` zurück, wenn der Text mehrzeilig ist, ansonsten wird `false` zurückgegeben.
`bool IsSingleLine() const;`	Gibt `true` zurück, wenn der Text nur einzeilig ist, ansonsten wird `false` zurückgegeben.

Tabelle 10.51 Die Methoden von »wxTextCtrl« (Forts.)

Methode	Beschreibung
`bool LoadFile( const wxString& filename );`	Lädt die Datei *filename* (wenn vorhanden) und zeigt diese im Textfeld an. Bei Erfolg wird `true`, ansonsten `false` zurückgegeben.
`void MarkDirty();`	Markiert den Text als verändert; Gegenstück zu `DiscardEdits`.
`virtual void Paste();`	Fügt den Text aus der Zwischenablage in das Textfeld ein.
`bool PositionToXY( long pos, long *x, long *y ) const;`	Konvertiert die angegebene Position `pos` in ein nullbasiertes Spalte-Zeile-Paar. In `x` finden Sie die Spalte und in `y` die Zeile wieder.
`virtual void Redo(),`	Wenn möglich (`CanRedo`), wird der zuvor ausgeführte Vorgang wiederholt.
`virtual void Remove( long from, long to );`	Löscht den Text von der Position `from` bis `to` (aber ohne das letzte Zeichen).
`virtual void Replace( long from, long to, const wxString& value );`	Ersetzt den Text von der Position `from` bis `to` durch den String `value`.
`bool SaveFile( const wxString& filename );`	Speichert den Inhalt des Textfelds in der Datei *filename*. Wenn alles glattgelaufen ist, wird `true`, ansonsten `false` zurückgegeben.
`bool SetDefaultStyle( const wxTextAttr& style);`	Verändert den Default-Stil und verwendet diesen beim nächsten neuen Text.
`virtual void SetEditable( const bool editable );`	Setzt das Textfeld auf editier- oder lesbar. Diese Methode überschreibt auch das `wxTE_READONLY`-Flag. Mit `true` setzen Sie den Text auf editierbar und mit `false` auf nur lesbar.
`virtual void SetInsertionPoint( long pos );`	Setzt den Einfügepunkt an die Position `pos`.
`virtual void SetInsertionPointEnd();`	Setzt den Einfügepunkt auf das Ende des Textes (gleichwertig zu `SetInsertionPoint(GetLastPosition())`).
`virtual void SetMaxLength( unsigned long len );`	Damit kann die maximale Anzahl von Zeichen (ohne `NULL`) vorgegeben werden, die der Anwender in das Textfeld schreiben kann. Versucht der Anwender mehr Zeichen einzugeben, als erlaubt, wird ein `wxEVT_COMMAND_TEXT_MAXLEN`-Ereignis generiert.
`virtual void SetSelection( long from, long to);`	Setzt den Text von `from` bis `to` als markiert. Wird für beide Parameter `–1` angegeben, wird der komplette Text ausgewählt.

Tabelle 10.51 Die Methoden von »wxTextCtrl« (Forts.)

Methode	Beschreibung
`bool SetStyle( long start, long end, const wxTextAttr& style );`	Ändert den Stil des Textes von der Position `start` bis `end` auf `style`.
`virtual void SetValue( const wxString& value );`	Setzt den Text im Textfeld auf `value`. Beachten Sie, dass hierbei nicht das `modfied`-Flag gesetzt wird. `IsModfied` gibt hierbei `false` zurück. Außerdem generiert diese Methode ein `wxEVT_COMMAND_TEXT_UPDATED`-Ereignis.
`void ShowPosition(long pos);`	Macht die Zeile mit der angegebenen Position sichtbar.
`virtual void Undo();`	Macht ein zuvor ausgeführtes Kommando rückgängig.
`void WriteText( const wxString& text);`	Schreibt den Text `text` in die aktuelle Einfügeposition des Textfelds.
`long XYToPosition( long x, long y ) ;`	Das Gegenstück zu `PositionToXY()`. Damit konvertieren Sie die Spalte (`x`) und die Zeile (`y`) wieder in eine Position. Bei einem Fehler wird `–1` zurückgegeben.
`wxTextCtrl& operator <<( const wxString& s) wxTextCtrl& operator <<(int i) wxTextCtrl& operator <<(long i) wxTextCtrl& operator <<(float f) wxTextCtrl& operator <<(double d) wxTextCtrl& operator <<(char c)`	der Ausgabeoperator (mehrfach überladen), der mit `wxTextCtrl` verwendet werden kann

Tabelle 10.51 Die Methoden von »wxTextCtrl« (Forts.)

Der Text wird immer mit `\n` als Zeilentrenner abgespeichert, unabhängig davon, ob Sie nun das Programm unter MS Windows, Linux/UNIX oder Mac OS schreiben. Allerdings gibt es hierbei einige Nachteile: Die Methode `GetValue()` zum Einlesen des kompletten Textes verwendet zum Beispiel auf Systemen, bei denen `\r\n` als Zeilentrenner (MS Windows beispielsweise) eingesetzt wird, diese Sequenzen. Dies bedeutet leider auch, dass Methoden wie `GetInsertionPoint`, `GetSelection`, `SetInsertionPoint` oder `SetSelection` nicht mit dem Indexwert arbeiten können. In der Dokumentation wird empfohlen, die Methode `GetRange` anstatt `GetValue` zu verwenden.

Leider bin ich mit diesem Hinweis auch nicht weitergekommen. In einem späteren Beispiel des Buches finden Sie einen Text-Editor, der eine Suchfunktion besitzt und den jeweils gefundenen Text markiert. Ich habe das Problem gelöst,

indem ich den Text zeilenweise eingelesen habe, was mir eigentlich viel effizienter erscheint, als den kompletten Text in den Speicher zu schieben, um dann nach einer String-Sequenz zu suchen.

`wxTextCtrl` generiert ebenfalls ein Ereignis der Klasse `wxCommandEvent`. Auf das Ereignis können Sie mit einem Handle folgendermaßen reagieren:

Handle einrichten	Beschreibung
`EVT_TEXT(id, func)`	Behandelt ein `wxEVT_COMMAND_TEXT_UPDATED`-Ereignis, das generiert wird, wenn der Text verändert wurde.
`EVT_TEXT_ENTER(id, func)`	Behandelt ein `wxEVT_COMMAND_TEXT_ENTER`-Ereignis, das generiert wird, wenn der Anwender die ↵-Taste gedrückt hat. Hier müssen Sie allerdings den Stil `wxTE_PROCESS_ENTER` beim Erzeugen verwenden.
`EVT_TEXT_MAXLEN(id, func)`	Behandelt ein `wxEVT_COMMAND_TEXT_MAXLEN`-Ereignis, das generiert wird, wenn der Anwender mehr Text eingibt, als mit `SetMaxLength` angegeben wurde (nur Windows und GTK+).

Tabelle 10.52 »wxSpinCtrl«-Ereignisse behandeln

Sie finden zwar jetzt ein kurzes Beispiel in unserem Listing zu `wxTextCtrl`, aber wir kommen darauf, wie bereits erwähnt, nochmals zurück, wenn wir einen Text-Editor erstellen. Das Panel mit dem Textfeld erzeugen wir wie immer mit

```
wxPanel* window9 = CreatePanelWithTextCtrl(choicebook);
```

Die komplette Funktion von `CreatePanelWithTextCtrl` sieht demnach folgendermaßen aus:

```
wxPanel *CreatePanelWithTextCtrl(wxChoicebook *parent) {
   wxPanel *panel = new wxPanel(parent, wxID_ANY);
   // Ein Text-Label in das Panel
   (void) new wxStaticText(panel, wxID_STATIC,
         wxT("Demonstriert die Klasse wxTextCtrl"),
         wxPoint(10, -1), wxDefaultSize, wxALIGN_LEFT);
   // Ein mehrzeiliges Textfeld erzeugen
   wxTextCtrl* text = new wxTextCtrl(
      panel, ID_TEXTCTRL, wxT("Hier den Text eingeben \n"),
      wxPoint( 10, 25 ), wxSize(300, 150),
      wxTE_MULTILINE|wxHSCROLL|wxTE_PROCESS_TAB );

   // auch dies ist möglich
   *text << wxT("float ") << 123.123 << wxT("\n");
   *text << wxT("Noch mehr Werte ") << 100 << wxT(" ")
          << wxT('A') << wxT("\n");
```

```
   return panel;
}
```

Nachdem Sie dieses Panel zum *Choicebook* hinzugefügt haben

```
choicebook->AddPage(window9, wxT("wxTextCtrl"), false, 8 );
```

ergibt sich folgendes Bild:

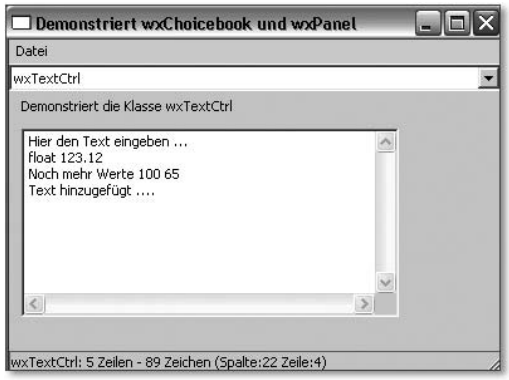

Abbildung 10.30 »wxTextCtrl« im Einsatz

Den Ereignis-Handle tragen wir wie folgt in der Tabelle ein:

```
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   ...
   EVT_TEXT(ID_TEXTCTRL, BasicFrame::OnText)
   ...
END_EVENT_TABLE()
```

Die Definition des Handles `OnText` sieht folgendermaßen aus:

```
// Textfeld auswerten
void BasicFrame::OnText( wxCommandEvent &event ) {
   wxTextCtrl *text = (wxTextCtrl*)
      FindWindowById(ID_TEXTCTRL);
   long x, y;
   text->PositionToXY( text->GetInsertionPoint(), &x, &y );
   wxString s = wxT("wxTextCtrl: ");
   s += wxString::Format(
      wxT("%d Zeilen - %d Zeichen (Spalte:%d Zeile:%d)"),
         text->GetNumberOfLines(), text->GetLastPosition(),
         x+1, y+1 );
   SetStatusText( s );
}
```

wxToggleButton

wxToggleButton ist ein Button, der gedrückt bleibt, wenn der Anwender ihn einmal gedrückt hat, ähnlich wie eine Checkbox, nur mit dem Aussehen eines wxButton.

Der Konstruktor von wxToggleButton:

```
wxToggleButton(
      wxWindow* parent,
      wxWindowID id,
      const wxString& label,
      const wxPoint& pos = wxDefaultPosition,
      const wxSize& size = wxDefaultSize,
      long style = 0,
      const wxValidator& val,
      const wxString& name = "checkBox" );
```

Die einzelnen Parameter haben folgende Bedeutung:

- parent – das Eltern-Fenster von wxToggleButton (sollte niemals NULL sein)
- id – die Identifizierung
- label – der Text auf dem Button
- pos – die Position; bei (–1, –1) wird wieder die Default-Position verwendet.
- size – die Größe; bei (–1, –1) wird wieder die Default-Größe verwendet.
- style – der Stil von wxToggleButton (hierbei gibt es keine speziellen Stile)
- validator – der Validator
- name – der Name für wxToggleButton

wxToggleButton besitzt im Grunde nur zwei direkte Methoden, die in Tabelle 10.53 kurz beschrieben werden.

Methode	Beschreibung
`bool GetValue() const;`	Ermittelt den Zustand des Toggle-Buttons. Wird `true` zurückgegeben, ist der Button gedrückt, ansonsten wird `false` zurückgegeben.
`void SetValue(const bool state);`	Versetzt den Toggle-Button in einen gedrückten (`state=true`) oder nicht gedrückten (`state=false`) Zustand.

Tabelle 10.53 Methoden für »wxToggleButton«

wxToggleButton generiert ein Ereignis der Klasse wxCommandEvent. Auf das Ereignis können Sie mit einem Handle folgendermaßen reagieren:

Handle einrichten	Beschreibung
`EVT_TOGGLEBUTTON(id, func)`	Behandelt ein `wxEVT_COMMAND_TOGGLEBUTTON_CLICKED`-Ereignis, das generiert wird, wenn der Anwender auf den Button klickt.

Tabelle 10.54 »wxToggleButton«-Ereignisse behandeln

Hierzu noch ein weiteres und letztes Beispiel in unserem *Choicebook*. Zunächst erzeugen Sie wieder ein Panel mit

```
wxPanel* window10 = CreatePanelWithToggleButton(choicebook);
```

Die komplette Funktion `CreatePanelWithToggleButton` dazu sieht folgendermaßen aus:

```
wxPanel *CreatePanelWithToggleButton(wxChoicebook *parent) {
   wxPanel *panel = new wxPanel(parent, wxID_ANY);
   // Ein Text-Label zum Panel
  (void) new wxStaticText(panel, wxID_STATIC,
        wxT("Demonstriert die Klasse wxToggleButton"),
        wxPoint(10, -1), wxDefaultSize, wxALIGN_LEFT);
   // Drei Toggle-Buttons erzeugen
   (void) new wxToggleButton(
         panel, ID_TOGGLE1, wxT("Mais"),
         wxPoint(10,20), wxDefaultSize);
   (void) new wxToggleButton(
         panel, ID_TOGGLE2, wxT("Bohnen"),
         wxPoint(10,50), wxDefaultSize);
   (void) new wxToggleButton(
         panel, ID_TOGGLE3, wxT("Erbsen"),
         wxPoint(10, 80), wxDefaultSize);
   return panel;
}
```

Jetzt fügen Sie das Panel zum *Choicebook* hinzu

```
choicebook->AddPage( window10,
                     wxT("wxToggleButton"), false, 9 );
```

und erhalten anschließend bei der Ausführung das in Abbildung 10.31 gezeigte Bild.

Hierzu noch das Einrichten des Ereignis-Handles in der Tabelle:

```
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   ...
   EVT_TOGGLEBUTTON(ID_TOGGLE1, BasicFrame::OnToggle)
   EVT_TOGGLEBUTTON(ID_TOGGLE2, BasicFrame::OnToggle)
```

```
   EVT_TOGGLEBUTTON(ID_TOGGLE3, BasicFrame::OnToggle)
END_EVENT_TABLE()
```

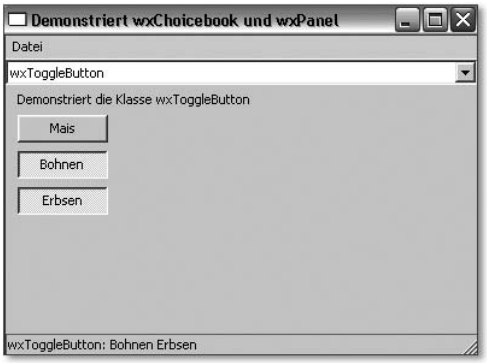

Abbildung 10.31 »wxToggleButton« bei der Ausführung

Und zuletzt noch die Definition des Handles:

```
// Toggle-Button auswerten
void BasicFrame::OnToggle( wxCommandEvent &event ) {
   wxToggleButton *t1 = (wxToggleButton*)
      FindWindowById(ID_TOGGLE1);
   wxToggleButton *t2 = (wxToggleButton*)
      FindWindowById(ID_TOGGLE2);
   wxToggleButton *t3 = (wxToggleButton*)
      FindWindowById(ID_TOGGLE3);
   wxString s = wxT("wxToggleButton: ");
   if( t1->GetValue() )
      s.Append(wxT("Mais "));
   if( t2->GetValue() )
      s.Append(wxT("Bohnen "));
   if( t3->GetValue() )
      s.Append(wxT("Erbsen "));
   SetStatusText( s );
}
```

Hinweis **[«]**

Bezüglich des kompletten Listings verweise ich Sie auf die Buch-CD, auf der Sie sämtliche Listings aus diesem Buch finden.

10.3.11 Statische Kontrollelemente

Statische Kontrollelemente sind im Grunde Informationen, die etwas anzeigen und nicht vom Anwender verändert werden können und somit auch keine Ereig-

nisse generieren. Da viele solcher statischen Elemente im letzten Listing verwendet wurden, folgt hier nur eine kurze Beschreibung zu den einzelnen Elementen.

[»] **Hinweis**

Wenn Sie die Bibliothek von `wxWidgets` heruntergeladen haben, finden Sie zu den statischen Kontrollelementen auch ein umfangreiches Beispiel.

wxGauge

`wxGauge` (siehe Abbildung 10.32) ist eine horizontale oder vertikale Leiste, die den aktuellen Fortschritt einer Arbeit anzeigt (häufig eine Zeitleiste). Für dieses Element werden keine Ereignisse generiert.

Abbildung 10.32 »wxGauge« zeigt den Zustand einer Arbeit an.

[»] **Hinweis**

Wer es gerne etwas komfortabler hat, dem sei das Widget `wxProgressDialog` empfohlen.

wxStaticText

Einen statischen Text haben Sie schon öfter eingesetzt. Damit können Sie an einer bestimmten Position einen Text mit einer oder mehreren nur lesbaren Zeilen anzeigen. Da dieses statische Element recht häufig verwendet wird, soll ein wenig näher darauf eingegangen werden. Der Konstruktor von `wxStaticText` ist folgendermaßen definiert:

```
wxStaticText( wxWindow* parent,
              wxWindowID id,
              const wxString& label,
              const wxPoint& pos = wxDefaultPosition,
              const wxSize& size = wxDefaultSize,
              long style = 0,
              const wxString& name = "staticText" );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster von `wxStaticText` (sollte niemals `NULL` sein)
- `id` – die Identifizierung; wird eigentlich nicht benötigt und kann auch mit `–1` oder `wxID_ANY` angegeben werden.
- `label` – der Text, der angezeigt werden soll
- `pos` – die Position; bei (–1, –1) wird wieder die Default-Position verwendet.

- `size` – die Größe; bei (–1, –1) wird wieder die Default-Größe verwendet.
- `style` – der Stil von `wxStaticText` (siehe Tabelle 10.55)
- `name` – der Name für `wxStaticText`

wxStaticText style	Beschreibung
`wxALIGN_LEFT`	Der Text ist links ausgerichtet.
`wxALIGN_RIGHT`	Der Text ist rechts ausgerichtet.
`wxALIGN_CENTRE`	Der Text wird zentriert ausgerichtet.
`wxST_NO_AUTORESIZE`	Gewöhnlich passt sich die Größe des Textes automatisch an, wenn beispielsweise `SetLabel` aufgerufen wird. Wollen Sie nicht, dass sich die Größe verändert, müssen Sie diesen Stil verwenden.

Tabelle 10.55 Stile von »wxStaticText«

Folgende zwei Methoden sind in der Klasse `wxStaticText` definiert:

Methode	Beschreibung
`wxString GetLabel() const;`	Gibt den Textinhalt zurück.
`virtual void SetLabel( const wxString& label );`	Setzt den statischen Text auf `label` und erneuert auch die Größenanpassung von `wxStaticText`, wenn bei der Erzeugung nicht das Flag `wxST_NO_AUTORESIZE` verwendet wurde.

Tabelle 10.56 Methoden von »wxStaticText«

wxStaticBitmap

`wxStaticBitmap` wird eingesetzt, um eine Bitmap anzuzeigen. Hierbei werden allerdings nicht Bilder und Grafiken angezeigt, sondern einfache Icons. Unter MS Windows ist die Größe der Bitmap auf 64×64 eingeschränkt. Eine einfache Bitmap auf einem Panel, wie im Abschnitt zuvor häufig verwendet, kann folgendermaßen angezeigt werden:

```
wxBitmap bitmap(wxT("Smile.bmp"), wxBITMAP_TYPE_BMP );
wxStaticBitmap* staticBitmap = new wxStaticBitmap(
   panel, wxID_STATIC, bitmap, wxPoint( 100, 20 ) );
```

Hierzu die Syntax des Konstruktors von `wxStaticBitmap`:

```
wxStaticBitmap(
       wxWindow* parent,
       wxWindowID id,
       const wxBitmap& label,
       const wxPoint& pos,
```

```
        const wxSize& size = wxDefaultSize,
        long style = 0,
        const wxString& name = "staticBitmap" );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster von `wxStaticBitmap` (sollte niemals `NULL` sein)
- `id` – die Identifizierung; wird eigentlich nicht benötigt und kann auch mit `–1` oder `wxID_ANY` angegeben werden.
- `label` – das Bitmap-Label
- `pos` – die Position; bei (–1, –1) wird wieder die Default-Position verwendet.
- `size` – die Größe; bei (–1, –1) wird wieder die Default-Größe verwendet.
- `style` – der Stil von `wxStaticBitmap`; hierfür sind keine speziellen Stile vorhanden.
- `name` – der Name für `wxStaticBitmap`

Folgende Methoden sind in `wxStaticBitmap` definiert:

Methode	Beschreibung
`wxBitmap` **`GetBitmap`**`() const;`	Gibt die Bitmap zurück, die aktuell verwendet wird.
`wxIcon` **`GetIcon`**`() const;`	Gibt das Icon zurück, das aktuell verwendet wird.
`virtual void` **`SetBitmap`**`( const wxBitmap& label );`	Setzt ein Bitmap-Label.
`virtual void` **`SetIcon`**`( const wxIcon& label );`	Setzt ein Icon-Label.

Tabelle 10.57 Methoden von »wxStaticBitmap«

wxStaticLine

`wxStaticLine` ist eine Linie, die zum Beispiel verwendet wird, um verschiedene Dialoge voneinander zu trennen. Diese Linie können Sie sowohl vertikal als auch horizontal einsetzen. Die Syntax des Konstruktors der Klasse `wxStaticLine` sieht folgendermaßen aus:

```
wxStaticLine(
      wxWindow* parent,
      wxWindowID id = wxID_ANY,
      const wxPoint& pos = wxDefaultPosition,
      const wxSize& size = wxDefaultSize,
      long style = wxLI_HORIZONTAL,
      const wxString& name = "staticLine" );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster von `wxStaticLine` (sollte niemals `NULL` sein)
- `id` – die Identifizierung; wird eigentlich nicht benötigt und kann auch mit `–1` oder `wxID_ANY` (Default) angegeben werden.
- `pos` – die Position; bei (–1, –1) wird wieder die Default-Position verwendet.
- `size` – die Länge und Breite der Linie; bei (–1, –1) wird wieder die Default-Größe verwendet.
- `style` – der Stil von `wxStaticLine`; hierfür kommt nur `wxLI_HORIZONTAL` für eine horizontale Linie und `wxLI_VERTICAL` für ein vertikale Linie in Frage.
- `name` – der Name für `wxStaticLine`

Für die Abfrage, ob es sich um eine vertikale Linie handelt, kann die Methode `IsVertical` verwendet werden. Gibt diese Methode `true` zurück, handelt es sich um eine vertikale Linie, bei `false` ist die Linie horizontal.

wxStaticBox

`wxStaticBox` ist ein rechteckiger Rahmen, der um andere Kontrollelemente gezeichnet werden kann, um eine logische Gruppe von Elementen zu markieren. Optional kann hierzu ein Text-Label verwendet werden.

Hinweis [«]

`wxStaticBox` haben Sie bereits im Abschnitt zuvor beim Listing zu `wxRadioButton` sehen können.

Man sollte niemals eine solche statische Box als Elternteil für weitere Elemente verwenden. Zudem sollten die Elemente in dieser statischen Box erst nach der Erzeugung der Box konstruiert werden (mit demselben Elternteil wie die statische Box!). Zwar soll sich dies in zukünftigen Versionen von `wxWidgets` ändern, aber im Augenblick sollten diese Dinge noch beachtet werden.

Hierzu noch die Syntax des Konstruktors von `wxStaticBox`:

```
wxStaticBox(
     wxWindow* parent,
     wxWindowID id,
     const wxString& label,
     const wxPoint& pos = wxDefaultPosition,
     const wxSize& size = wxDefaultSize,
     long style = 0,
     const wxString& name = "staticBox" );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster von `wxStaticBox` (sollte niemals `NULL` sein)
- `id` – die Identifizierung; wird eigentlich nicht benötigt und kann auch mit `–1` oder `wxID_ANY` (Default) angegeben werden.
- `label` – das Text-Label der statischen Box
- `pos` – die Position; bei (–1, –1) wird wieder die Default-Position verwendet.
- `size` – die Länge und Breite der Box; bei (–1, –1) wird wieder die Default-Größe verwendet.
- `style` – der Stil von `wxStaticBox`; hierfür gibt es aktuell keine speziellen Stile.
- `name` – der Name für `wxStaticBox`

`wxStaticBox` hat außerdem auch keine speziellen Methoden definiert.

10.3.12 Menüs

Jetzt sollen endlich die Menüs (`wxMenu`) beschrieben werden, von denen Sie schon regen Gebrauch gemacht haben. Mittlerweile haben Sie selbst festgestellt, dass solche Menüs dazu eingesetzt werden, um mehrere Kommandos auf kleinstem Raum zu verwenden und anzuzeigen. Ich denke, ich muss nichts über Menüs erzählen, selbst ein wenig computerorientierter Anwender kommt damit schnell zurecht. In diesem Abschnitt geht es darum, wie Sie dem Anwender selbst ein solches Menü zur Verfügung stellen können.

wxMenu

Ein Menü listet viele Kommandos auf, die entweder in einer Menübar oder in einem Kontextmenü erscheinen. Das Kontextmenü wird in der Regel aufgerufen, wenn Sie in einem bestimmten Bereich die rechte Maustaste drücken. Abgesehen von den normalen Elementen kann man in einem solchen Menü auch Untermenüs, Radio- und Check-Buttons neben dem Text-Label platzieren. Ein Menüelement kann man selbstverständlich auch deaktivieren, so dass die Schrift grau erscheint und das Element nicht anwählbar ist.

Der Konstruktor von `wxMenu` hat folgende Syntax:

```
wxMenu(const wxString& title = "", long style = 0):
```

Die Parameter haben folgende Bedeutung:

- `title` – der Titel für ein Popup-Menü (Kontextmenü), falls eines verwendet wird. Wird hier nichts angegeben, gibt es keinen Titel (Default).
- `style` – geben Sie hierbei `wxMENU_TEAROFF` an, ist das Menü ablösbar (nur bei `wxGTK` möglich).

wxMenu besitzt viele wichtige Methoden, die in Tabelle 10.58 aufgelistet sind und näher beschrieben werden.

Methode	Beschreibung
`wxMenuItem* Append( int id, const wxString& item, const wxString& helpString="", wxItemKind kind=wxITEM_NORMAL);`	Fügt ein String-Element an das Ende des Menüs. id ist der Identifizierer für das Menükommando. item ist der Text, der im Menü angezeigt wird, helpString kann optional verwendet werden und wird gegebenenfalls vom Ereignis-Handle wxEVT_MENU_HIGHLIGHT standardmäßig in der Statusleiste angezeigt. Für kind kann wxITEM_SEPARATOR, wxITEM_NORMAL, wxITEM_CHECK oder wxITEM_RADIO angegeben werden. Mehr zum item-String entnehmen Sie dem Ende dieser Tabelle.
`wxMenuItem* Append( int id, const wxString& item, wxMenu *subMenu, const wxString& helpString="");`	Fügt ein nach rechts ausfahrbares Untermenü am Ende des Menüs hinzu. Dieses Untermenü sollten Sie jedoch erst hinzufügen, nachdem Sie die Menüelemente für das Untermenü hinzugefügt haben. Das Untermenü wird mit dem Parameter subMenu angegeben. Der Rest entspricht der letzten Version (ohne kind). Mehr zum item-String entnehmen Sie dem Ende dieser Tabelle.
`wxMenuItem* Append( wxMenuItem* menuItem);`	Damit können Sie ein komplettes wxMenuItem-Objekt dem Menü hinzufügen. Zunächst erscheint dies ein wenig aufwendiger, aber mit wxMenuItem kann man einzelne Menüelemente noch viel flexibler (bezüglich der Optik) behandeln. Im anschließenden Beispiel finden Sie beispielsweise zwei Menüelemente mit einer Bitmap.
`wxMenuItem* AppendCheckItem( int id, const wxString& item, const wxString& helpString="");`	Fügt ein Menüelement mit Check-Häkchen an das Ende des Menüs. Zur Bedeutung der einzelnen Parameter siehe Append.
`wxMenuItem* AppendRadioItem( int id, const wxString& item, const wxString& helpString="");`	Fügt ein Radio-Menüelement an das Ende des Menüs. Es kann immer nur ein Radio-Element im Menü aktiviert sein. Diese Methode ist derzeit nur unter MS Windows und GTK vorhanden. Wollen Sie zur Übersetzung überprüfen, ob diese Radio-Menüs vorhanden sind, müssen Sie nur eine bedingte Kompilierung mit #if wxHAS_RADIO_MENU_ITEMS machen. Zur Bedeutung der einzelnen Parameter siehe wieder Append.

Tabelle 10.58 Methoden von »wxMenu«

Methode	Beschreibung
`wxMenuItem* AppendSeparator();`	Fügt einen Separator (eine Trennlinie) am Ende des Menüs hinzu.
`void Break();`	Fügt eine Trennung in das Menü, so dass das nächste Element in einer neuen Spalte angezeigt wird.
`void Check( int id, const bool check );`	Setzt oder entfernt das Häkchen vor einem Check-Menüelement mit der ID `id`. Mit `false` für `check` entfernen Sie das Häkchen und mit `true` setzen Sie es.
`void Delete(int id);`	Entfernt ein Element des Menüs mit der ID `id`. Gilt nicht für Elemente eines Untermenüs.
`void Delete(wxMenuItem *item);`	Entfernt das Element `item` (ein `wxMenuItem`-Objekt). Gilt nicht für Elemente eines Untermenüs.
`void Destroy(int id);`	Entfernt ein Element des Menüs mit der ID `id`. Gilt jetzt auch für Elemente eines Untermenüs, nicht aber für den Halter des Untermenüs (siehe `Remove`).
`void Destroy(wxMenuItem *item);`	Entfernt das Element `item` (ein `wxMenuItem`-Objekt). Gilt jetzt auch für Elemente eines Untermenüs, nicht aber für den Halter des Untermenüs (siehe `Remove`).
`void Enable( int id, const bool enable );`	Schaltet das Menüelement mit der ID `id` ein (`enable=true`) oder aus (`enable=false`), es wird ausgegraut.
`int FindItem( const wxString& itemString) const;`	Sucht nach einem Menüelement mit dem String `itemString`. Wird kein entsprechendes Element gefunden, wird `wxNOT_FOUND` zurückgegeben.
`wxMenuItem* FindItemByPosition( size_t position) const;`	Gibt ein `wxMenuItem` zurück, das sich an der Position `position` befindet.
`wxString GetHelpString( int id ) const;`	Gibt den Hilfs-String der ID `id` zurück, der gewöhnlich in der Statuszeile angezeigt wird.
`wxString GetLabel(int id) const;`	Gibt das Text-Label des Menüelements mit der ID `id` zurück.
`size_t GetMenuItemCount() const;`	Gibt die Anzahl der Elemente eines Menüs zurück.
`wxString GetTitle() const;`	Gibt den Titel-String des Menüs zurück. Dies macht gewöhnlich Sinn bei Popup-Menüs.

Tabelle 10.58 Methoden von »wxMenu« (Forts.)

Methode	Beschreibung
`wxMenuItem* Insert( size_t pos, wxMenuItem *item );` `wxMenuItem* Insert( size_t pos, int id, const wxString& item, const wxString& helpString="", wxItemKind kind=wxITEM_NORMAL);`	Fügt ein Menüelement an der Position `pos` hinzu. Die restlichen Parameter sind gleichwertig zur entsprechenden `Append`-Methode.
`wxMenuItem* InsertCheckItem( size_t pos, int id, const wxString& item, const wxString& helpString="");`	Fügt ein Check-Menüelement an der Position `pos` hinzu. Die restlichen Parameter sind gleichwertig zur entsprechenden `AppendCheckItem`-Methode.
`wxMenuItem* InsertRadioItem( size_t pos, int id, const wxString& item, const wxString& helpString="");`	Fügt ein Radio-Menüelement an der Position `pos` hinzu. Die restlichen Parameter sind gleichwertig zur entsprechenden `AppendRadioItem`-Methode.
`wxMenuItem* InsertSeparator( size_t pos );`	Fügt einen Separator (Trennlinie) an der Position `pos` im Menü ein.
`bool IsChecked(int id) const;`	Überprüft, ob das Check-Menüelement mit der ID `id` ein Häkchen hat (`true`) oder nicht (`false`).
`bool IsEnabled(int id) const;`	Überprüft, ob das Menüelement mit der ID `id` aktiviert (`true`) oder deaktiviert (`false`) ist (deaktiviert ist gewöhnlich ausgegraut, siehe auch die Methode `Enable`).
`wxMenuItem* Prepend ( wxMenuItem *item );` `wxMenuItem* Prepend ( int id, const wxString& item, const wxString& helpString="", wxItemKind kind=wxITEM_NORMAL);`	Fügt ein Menüelement am Anfang (Position `0`) hinzu. Die restlichen Parameter sind gleichwertig zur entsprechenden `Append`-Methode.
`wxMenuItem* PrependCheckItem( int id, const wxString& item, const wxString& helpString="");`	Fügt ein Check-Menüelement am Anfang (Position `0`) hinzu. Die restlichen Parameter sind gleichwertig zur entsprechenden `AppendCheckItem`-Methode.
`wxMenuItem* PrependRadioItem( int id, const wxString& item, const wxString& helpString="");`	Fügt ein Radio-Menüelement am Anfang (Position `0`) hinzu. Die restlichen Parameter sind gleichwertig zur entsprechenden `AppendRadioItem`-Methode.

Tabelle 10.58 Methoden von »wxMenu« (Forts.)

Methode	Beschreibung
`wxMenuItem* PrependSeparator();`	Fügt einen Separator (Trennlinie) am Anfang (Position `0`) hinzu.

`wxMenuItem * Remove(int id);`
`wxMenuItem * Remove(`
`wxMenuItem *item );`
	Entfernt das Menüelement mit der ID `id` oder einem `wxMenuItem`-Objekt aus dem Menü, löscht aber nicht das Objekt. Damit können Sie praktisch einzelne Elemente entfernen und später wieder hinzufügen.

`void SetHelpString(`
`int id,`
`const wxString& helpString );`
	Setzt den Hilfs-String des Menüelements mit der ID `id`. Der Hilfs-String wird gewöhnlich in der Statusleiste angezeigt.

`void SetLabel(`
`int id,`
`const wxString& label );`
	Setzt das Text-Label für das Menüelement mit der ID `id` auf `label`.

`void SetTitle(` `const wxString& title );`	Setzt den Titel des Menüs. Macht nur Sinn bei einem Popup-Menü (Kontextmenü).

`void UpdateUI(`
`wxEvtHandle* source = NULL)`
`const;`
	Sendet das Ereignis, um ein Menü-UI zu erneuern. Diese Methode kann beispielsweise aufgerufen werden, wenn ein Menü mit `wxWindow::PopupMenu` aufgeklappt wird.

Tabelle 10.58 Methoden von »wxMenu« (Forts.)

[»] Hinweis

Es war die Rede von `wxMenuItem`, wozu Sie im Listing auch ein Beispiel finden, auch wenn wir darauf nicht mehr genauer eingehen. Mit `wxMenuItem` kann man recht komfortabel das Aussehen eines einzelnen Menüelements (Hintergrundfarbe, Bitmap, Schriftart, Ausrichtung etc.) beeinflussen.

Der `item`-String der Methode `Append` (bzw. wenn ein `wxItemMenu`-Objekt verwendet wird) kann auch Tastenkürzel enthalten, mit denen Sie die einzelnen Menüelemente über die Tastatur aktivieren können. Wenn Sie eine solche Tastenkombination verwenden wollen, müssen Sie nur hinter dem `item`-String, getrennt durch ein `\t`, die Tastenkombination angeben. Wollen Sie beispielsweise die Kombination Ctrl+Q verwenden, sieht der `item`-String folgendermaßen aus:

```
"Beenden\tCtrl-Q"
```

Dabei ist es irrelevant, ob Sie »CTRL« oder »Ctrl« schreiben – es wird nicht auf Groß- und Kleinschreibung geachtet. Außerdem können Sie anstelle des Minuszeichens zwischen »Ctrl« und »Q« auch ein Pluszeichen setzen. Natürlich können Sie neben Ctrl noch viele weitere Tasten und Tastenkombinationen verwenden. So stehen Ihnen zum Beispiel auch mit F1 bis F12 alle Funktionstasten

zur Verfügung. Wenn Ihnen die Tastenkombinationen ausgehen, können Sie noch weitere Tasten kombinieren. Beispielsweise muss mit

```
"Beenden\tCtrl-Shift-Q"
```

für die Aktivierung des Kommandos BEENDEN die Tastenkombination Ctrl und ⇧ und Q gedrückt werden. In Tabelle 10.59 finden Sie einen Überblick über gängige Tasten (auch hier wird zwischen Groß- und Kleinschreibung nicht unterschieden).

Taste	Beschreibung
DEL oder DELETE	Delete-Taste
INS oder INSERT	Insert-Taste
ENTER oder RETURN	Enter-Taste
PGUP	PageUp-Taste
PGDN	PageDown-Taste
LEFT	Pfeil-nach-links-Taste
RIGHT	Pfeil-nach-rechts-Taste
UP	Pfeil-nach-oben-Taste
DOWN	Pfeil-nach-unten-Taste
HOME	Home-Taste
END	End-Taste
SPACE	Leertaste
TAB	Tabulator-Taste
ESC oder ESCAPE	Escape-Taste (nur MS Windows)
SHIFT	Shift-Taste
ALT	Alt-Taste

Tabelle 10.59 Tastenkürzel

Mit `wxMenu` sind mehrere Arten von Ereignissen verbunden: `wxCommandEvent`, `wxUpdateUIEvent`, `wxMenuEvent` und `wxContextMenuEvent`.

Die in Tabelle 10.60 aufgelisteten Makros für das Einrichten eines Ereignis-Handles werden für ein `wxCommandEvent`-Argument eingesetzt (was wohl der häufigste Fall sein dürfte). Diese Makros lassen sich auch auf die noch kommende Klasse `wxToolBar` und die Popup-Menüs anwenden. So können Sie ein Menüelement, ein Toolbar-Element und sogar ein Kontextmenü-Element mit demselben Handle verwenden (siehe auch das anschließende Listing).

Handle einrichten	Beschreibung
`EVT_MENU(id, func)`	Behandelt ein `wxEVT_COMMAND_MENU_SELECTED`-Ereignis, das generiert wird, wenn ein Menüelement ausgewählt wurde.
`EVT_MENU_RANGE(id1, id2, func)`	Behandelt ein `wxEVT_COMMAND_MENU_RANGE`-Ereignis, das generiert wird, wenn ein Menüelement zwischen dem Bereich `id1` und `id2` ausgewählt wurde.

Tabelle 10.60 »wxMenu«-Ereignisse behandeln (»wxCommandEvent«)

In Tabelle 10.61 finden Sie die Ereignis-Makros, die verwendet werden, wenn der Ereignis-Handle ein `wxUpdateUIEvent` als Argument verwendet. Gewöhnlich wird ein solches Ereignis in einer Leerlaufzeit (engl.: *idle time*) des Programms generiert, um der Anwendung die Möglichkeit zu geben, das UI (User Interface) zu erneuern, zum Beispiel, um ein Menüelement zu aktivieren oder zu deaktivieren. Sie können ein solches Ereignis auch von Hand mit der Methode `UpdateUI` auslösen.

Handle einrichten	Beschreibung
`EVT_UPDATE_UI(id, func)`	Behandelt ein `wxEVT_UPDATE_UI`-Ereignis. Im Handle haben Sie nun die Möglichkeit, einzelne Menüelemente zu aktivieren oder zu deaktivieren bzw. zu verändern. Hierzu stehen die Methoden `Check`, `Enable`, `SetText` und noch weitere zur Verfügung.
`EVT_UPDATE_UI_RANGE( id1, id2, func )`	Behandelt ein Ereignis `wxEVT_UPDATE_UI`. Im Handle können Sie nun dasselbe wie mit `EVT_UPDATE_UI` machen, nur gilt dies jetzt für die Menüelemente aus dem Bereich `id1` und `id2`.

Tabelle 10.61 »wxMenu«-Ereignisse behandeln (»wxUpdateUI«)

[»] **Hinweis**

Wenn Sie Informationen zu weiteren Methoden von `wxUpdateUIEvent` benötigen, sollten Sie die Dokumentation zur Hand nehmen (siehe Buch-CD).

Wollen Sie hingegen die Ereignisse des Kontextmenüs mit einem Handle behandeln (mit `wxContextMenuEvent` als Argument), finden Sie auch hierzu zwei Makros zum Einrichten eines Ereignis-Handles (siehe Tabelle 10.62). Ein Kontextmenü wird im Allgemeinen mit einem rechten Mausklick ausgefahren. So können Sie zum Beispiel mit `wxContextMenuEvent` und der darin enthaltenen Methode `GetPosition` ermitteln, an welcher Position das Kontextmenü aufgeklappt wurde.

Handle einrichten	Beschreibung
`EVT_CONTEXT_MENU(func)`	Wird generiert, wenn der Anwender ein Popup-Menü angefordert hat, beispielsweise über einen rechten Mausklick.
`EVT_COMMAND_CONTEXT_MENU(id, func)`	Wie `EVT_CONTEXT_MENU`, nur wird hierbei ein Fenster-Identifizierer verwendet (beispielsweise bei mehreren Kind-Fenstern oder *Notebooks*).

Tabelle 10.62 »wxMenu«-Ereignisse behandeln (»wxContextMenuEvent«)

Zu guter Letzt haben Sie noch die Klasse `wxMenuEvent`, die alle menütypischen Ereignisse verarbeitet. Auch hier gibt es eine ganze Reihe von Makros, um einen Handle einzurichten:

Handle einrichten	Beschreibung
`EVT_MENU_OPEN(func)`	Behandelt das `wxEVT_MENU_OPEN`-Ereignis, das generiert wird, wenn ein Menü gerade geöffnet (ausgefahren) wird.
`EVT_MENU_CLOSE(func)`	Behandelt das `wxEVT_MENU_CLOSE`-Ereignis, das generiert wird, wenn ein Menü gerade geschlossen wird.
`EVT_MENU_HIGHLIGHT(id, func)`	Behandelt das `wxEVT_MENU_HIGHLIGHT`-Ereignis, das generiert wird, wenn ein Menüelement mit ID `id` hervorgehoben (Mouseover) wird. Standardmäßig wird dieses Ereignis verwendet, um in der Statusleiste einen Text anzuzeigen.
`EVT_MENU_HIGHLIGHT_ALL(func)`	Wie `EVT_MENU_HIGHLIGHT`, nur gilt dies für alle Menü-Identifizierer.

Tabelle 10.63 »wxMenu«-Ereignisse behandeln (»wxMenuEvent«)

Hinweis **[«]**

Ein Beispiel zu `wxMenu` finden Sie im übernächsten Abschnitt, »wxMenuBar«.

Kontrollbalken

Es gibt drei verschiedene Arten von Kontrollbalken: `wxMenuBar`, `wxToolBar` und `wxStatusBar`. `wxMenuBar` kann nur in Verbindung mit einem `wxFrame` verwendet werden. `wxToolBar` und `wxStatusBar` sind zwar nicht an ein `wxFrame` gebunden, werden aber gewöhnlich nur in Verbindung damit eingesetzt.

wxMenuBar

Ein Menübalken beginnt im Allgemeinen an der linken oberen Seite eines Frames und enthält eine ganze Reihe von Menüelementen (beispielsweise `wxMenu`, siehe Abbildung 10.33).

Abbildung 10.33 wxMenuBar – »Datei«

In der Praxis erzeugt man zunächst ein `wxMenuBar`-Objekt. Anschließend wird noch ein `wxMenu`-Objekt (siehe Abschnitt »wxMenu«) erzeugt, in das die einzelnen Menüelemente hinzugefügt werden. Am Ende fügen Sie das `wxMenu`-Objekt mit seinen einzelnen Menüelementen dem Menübalken (`wxMenuBar`) hinzu – fertig ist das Menü. Hierzu die Syntax des Konstruktors von `wxMenuBar`:

```
// Standard-Konstruktor
wxMenuBar(long style = 0);
// Erzeugt eine Menübar von einem Array von Menüs und Titeln
wxMenuBar( size_t n,_
           wxMenu* menus[],
           const wxString titles[],
           long style = 0 );
```

In der Praxis wird hierbei meistens der Standardkonstruktor verwendet. Trotzdem soll auch auf die einzelnen Parameter der beiden Konstruktoren eingegangen werden:

- `n` – Anzahl der Menüs
- `menu` – ein Array von Menüs
- `titles` – ein Array von Titel-Strings
- `style` – hierbei können Sie mit `wxMB_DOCKABLE` (nur `wxGTK`) den Menübalken loslösen.

Auch in der Klasse `wxMenuBar` sind einige nützliche Methoden (ähnlich wie die von `wxMenu`) definiert. In Tabelle 10.64 sind die wichtigsten aufgelistet.

Methode	Beschreibung

`bool` **Append**`(`
`wxMenu *menu,`
`const wxString& title );`
	Fügt ein `wxMenu`-Objekt (`menu`) am Ende des Menübalkens ein. Den Titel des Menübalkens geben Sie mit `title` an.

Tabelle 10.64 Wichtige Methoden von »wxMenuBar«

Methode	Beschreibung
`void EnableTop( int pos, const bool enable );`	Aktiviert oder deaktiviert ein ganzes Menü mit der Position `pos` des Menübalkens. Mit `false` wird das Menü deaktivert und mit `true` wieder aktiviert. Diese Methode funktioniert allerdings nur, wenn der Menübalken mit dem Frame angeschlossen ist. Dies ist der Fall, wenn die Methode `wxFrame::SetMenuBar` verwendet wird.
`int FindMenu( const wxString& title) const;`	Gibt den Index des Menüs zurück, dessen Titel `title` ist, oder, wenn kein entsprechender Menübalken vorhanden ist, `wxNOT_FOUND`. Der `title`-Parameter kann entweder der Menütitel oder ein Menü-Label sein.
`int FindMenuItem( const wxString& menuString, const wxString& itemString) const;`	Findet die ID für ein Menüelement, das dem String-Paar `menuString` (Menütitel) und `itemString` (Menüelement) entspricht, oder `wxNOT_FOUND`, wenn nichts gefunden wurde.
`bool Insert( size_t pos, wxMenu *menu, const wxString& title );`	Wie `Append`, nur wird das Menü an der Position `pos` des Menübalkens eingefügt.
`void Refresh();`	Zeichnet die Menübalken neu.
`wxMenu *Remove(size_t pos);`	Entfernt das komplette Menü an der Position `pos` vom Menübalken. Da das Menüobjekt dabei zurückgegeben wird, kann es jederzeit wieder eingefügt werden (gewöhnlich mit `Insert`).
`wxMenu *Replace( size_t pos, wxMenu *menu, const wxString& title );`	Ersetzt das komplette Menü an der Position `pos` durch das Menü `menu` und verwendet als Titel `title`. Zurückgegeben wird das ersetzte Menü von der Position `pos`.

Tabelle 10.64 Wichtige Methoden von »wxMenuBar« (Forts.)

Alles, was Sie über die Ereignisse und deren Behandlung im Abschnitt zu `wxMenu` gelesen haben, gilt auch für `wxMenuBar`.

Bevor Sie nun endlich ein Beispiel in der Praxis sehen, werden noch kurz die Toolbar und das Popup-Menü beschrieben, die ähnliche Eigenschaften und Bedeutungen wie die Menüs haben.

wxToolBar

Eine Toolbar kann mehrere Buttons oder auch andere Kontrollelemente (wie zum Beispiel eine Combo-Box) enthalten (siehe Abbildung 10.34). Die Anordnung kann horizontal oder vertikal erfolgen. Natürlich kann man hierbei auch Check-Buttons (oder häufig auch Toggle-Buttons) oder Radio-Buttons verwen-

den. Mit dem Toolbar-Button können Sie sowohl Text-Labels als auch Bitmaps anzeigen lassen. Wenn Sie eine Toolbar mit den Methoden `wxFrame::CreateToolBar` oder `wxFrame::SetToolBar` erzeugen, wird das Frame dieses Objekt verwalten. Sollten Sie die Toolbar anders anwenden wollen, müssen Sie den Code für die Größe und die Position der Toolbar selbst in die Hand nehmen.

Abbildung 10.34 »wxToolBar« mit Icons und einer Combo-Box

Die Syntax des Konstruktors zum Erzeugen einer `wxToolBar` sieht folgendermaßen aus:

```
wxToolBar( wxWindow* parent,
           wxWindowID id,
           const wxPoint& pos = wxDefaultPosition,
           const wxSize& size = wxDefaultSize,
           long style = wxTB_HORIZONTAL | wxNO_BORDER,
           const wxString& name = wxPanelNameStr );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster der Toolbar
- `id` – die Identifizierung der Toolbar
- `pos` – die Position der Toolbar; bei (–1, –1) wird wieder die Default-Position verwendet.
- `size` – die Größe der Toolbar; bei (–1, –1) wird wieder die Default-Größe verwendet.
- `style` – der Stil der Toolbar (siehe Tabelle 10.65)
- `name` – der Name

Stil (wxToolBar)	Beschreibung
`wxTB_HORIZONTAL`	Erzeugt eine horizontale Toolbar.
`wxTB_VERTICAL`	Erzeugt eine vertikale Toolbar.
`wxTB_FLAT`	Gibt der Toolbar ein flaches Aussehen (nur MS Windows und GTK+).
`wxTB_DOCKABLE`	Damit kann man die Position der Toolbar verändern oder frei platzieren (bisher nur GTK+).
`wxTB_TEXT`	Zeigt den Text in den Toolbar-Buttons an. Normalerweise wird hier ein Icon angezeigt.

Tabelle 10.65 Die Stile für »wxToolBar«

Stil (wxToolBar)	Beschreibung
wxTB_NOICONS	Zeigt keine Icons in den Toolbar-Buttons an.
wxTB_NODIVIDER	Verwendet keinen Teiler (Trennlinie) oberhalb der Toolbar (nur MS Windows).
wxTB_HORZ_LAYOUT	Zeigt einen Text und Icons an (nur MS Windows und GTK+). Zu diesem Stil muss wxTB_TEXT verwendet werden.
wxTB_HORZ_TEXT	Kombiniert wxTB_HORZ_LAYOUT und wxTB_TEXT.

Tabelle 10.65 Die Stile für »wxToolBar« (Forts.)

Nachdem Sie eine Toolbar mit dem Konstruktor erzeugt haben, können Sie die Methode `wxToolBar::AddTool` verwenden. Anschließend müssen Sie die Methode `wxToolBar::Realize` einsetzen, um die Toolbar zu konstruieren und anzuzeigen.

Wenn Sie wollen, dass das Frame diese Toolbar verwaltet, sollten Sie die Methode `wxFrame::SetToolBar` (oder auch `wxFrame::CreateToolBar`) verwenden. Auch die Klasse `wxToolBar` bietet viele Methoden an, von denen Sie wieder die wichtigsten in Tabelle 10.66 finden.

Methode	Beschreibung
`bool AddControl( wxControl* control );`	Fügt ein Kontrollelement zur Toolbar hinzu (zum Beispiel eine Combo-Box).
`void AddSeparator();`	Fügt eine Trennlinie zur Toolbar hinzu.
`wxToolBarToolBase* AddTool( int toolId, const wxBitmap& bitmap1, const wxString& shortHelpString = "", wxItemKind kind=wxITEM_NORMAL);`	Fügt ein Tool zur Toolbar hinzu. Mit `toolId` geben Sie die ID des Elements an. Das Icon geben Sie mit `bitmap1` an und der Text, der erscheint, wenn Sie mit der Maus über das Icon fahren (auch als Tooltipp bekannt), wird mit `shortHelpString` angegeben. Mit `kind` stehen Ihnen folgende Buttons zur Verfügung: ein normaler Button (`wxITEM_NORMAL`), ein Button zum »Abhaken« (Check-Button oder Toggle-Button; `wxITEM_CHECK`), der gedrückt bleibt, wenn er betätigt wurde, und ein Radio-Button (`wxITEM_RADIO`), bei dem immer nur ein Button aus einer Gruppe von Buttons aktiviert ist.

Tabelle 10.66 Einige Methoden von »wxToolBar«

Methode	Beschreibung
`wxToolBarToolBase* AddTool( int toolId, const wxBitmap& bitmap1, const wxBitmap& bitmap2 = wxNullBitmap, wxItemKind kind=wxITEM_NORMAL, const wxString& shortHelpString = "", const wxString& longHelpString = "", wxObject* clientData = NULL );`	Fügt ein Tool zur Toolbar hinzu. Mit `toolId` geben Sie die ID des Elements an, das Icon geben Sie mit `bitmap1` an. Der Text, der erscheint, wenn Sie mit der Maus über das Icon fahren (auch als Tooltipp bekannt), wird mit `shortHelpString` angegeben. Das `bitmap2` wird verwendet, wenn Sie den Button deaktivieren. Gewöhnlich wird dieser ausgegraut, aber es kann hier auch eine Bitmap verwendet werden. Mit `kind` stehen Ihnen folgende Buttons zur Verfügung: ein normaler Button (`wxITEM_NORMAL`), ein Button zum »Abhaken« (Check-Button oder Toggle-Button; `wxITEM_CHECK`), der gedrückt bleibt, wenn er betätigt wurde, und ein Radio-Button (`wxITEM_RADIO`), bei dem immer nur ein Button aus einer Gruppe von Buttons aktiviert ist. Mit `longHelpString` können Sie zusätzlich zu `shortHelpString` weitere Informationen angeben, die in der Statusleiste angezeigt werden (falls eine vorhanden ist). Optional können Sie auch wieder Client-Daten verwenden, die Sie später mit der Methode `wxToolBar::GetToolClientData` erneut einsetzen können. Wenn Sie die einzelnen Tools zur Toolbar hinzugefügt haben, müssen Sie die Methode `wxToolBar::Realize` aufrufen.
`wxToolBarToolBase* AddTool( wxToolBarToolBase* tool );`	Hiermit können Sie ein komplett fertiges Tool zur Werkzeugleiste hinzufügen.
`bool DeleteTool(int toolId);`	Damit können Sie ein Tool mit der ID `id` von der Toolbar entfernen und löschen. Hierbei ist es auch nicht nötig, dass Sie die Methode `Realize` aufrufen, weil die Veränderung auf der Stelle vorgenommen wird.
`bool DeleteToolByPos( size_t pos );`	Wie die Methode `DeleteTool`, nur können Sie hierbei ein Tool an einer bestimmten Position statt mit einer bestimmten ID entfernen.
`void EnableTool( int toolId, const bool enable );`	Damit können Sie ein Tool mit der ID `toolId` deaktivieren und aktivieren. Deaktiviert wird ein Tool, wenn Sie für `enable` `false` angeben, und aktiviert wird ein Tool, wenn Sie `true` für `enable` angeben. Diese Methode kann allerdings erst nach dem Aufruf der Methode `Realize` eingesetzt werden.
`wxToolBarToolBase* FindById( int id );`	Sucht ein Tool mit der ID `id` und gibt dieses zurück. Falls kein entsprechendes Tool gefunden wird, wird `NULL` zurückgegeben.

Tabelle 10.66 Einige Methoden von »wxToolBar« (Forts.)

Methode	Beschreibung
`wxToolBarToolBase * InsertControl( size_t pos, wxControl *control );`	Fügt ein Kontrollelement zur Toolbar an der Position `pos` hinzu (zum Beispiel eine Combo-Box). Auch hierbei muss wieder die Methode `Realize` aufgerufen werden, um die Änderungen anzuzeigen.
`wxToolBarToolBase * InsertSeparator( size_t pos);`	Fügt eine Trennlinie an der Position `pos` zur Toolbar hinzu. Anschließend muss wieder die Methode `Realize` aufgerufen werden, um die Veränderungen sichtbar zu machen.
`wxToolBarToolBase * InsertTool(size_t pos, int toolId, const wxBitmap& bitmap1, const wxBitmap& bitmap2 = wxNullBitmap, bool isToggle = false, wxObject* clientData = NULL, const wxString& shortHelpString = "", const wxString& longHelpString = ""); wxToolBarToolBase * InsertTool(size_t pos, wxToolBarToolBase* tool);`	Gleichwertig zu denselben Parametern bei `AddTool`, nur kann hiermit ein Tool an einer bestimmten Position (1. Parameter) eingefügt werden. Auch hierbei muss wieder die Methode `Realize` aufgerufen werden, um die Änderungen anzuzeigen.
`bool Realize();`	Diese Methode sollte immer aufgerufen werden, wenn Sie ein neues Tool zur Toolbar hinzugefügt haben.
`wxToolBarToolBase *RemoveTool( int id );`	Entfernt das Tool mit der ID `id` von der Toolbar, um das Speicher-Objekt zu löschen. Das Entfernen bezieht sich nur auf den visuellen Aspekt. Als Rückgabewert erhalten Sie einen Zeiger auf das entfernte Tool und können es somit wieder in diese oder eine andere Toolbar einfügen.

Tabelle 10.66 Einige Methoden von »wxToolBar« (Forts.)

Die Makros zum Einrichten eines Ereignis-Handles für Toolbars finden Sie in Tabelle 10.67. Die Klasse `wxToolBar` erhält ihre Ereignisse auf demselben Weg, wie es schon bei `wxMenuBar` in einem Frame der Fall war und beschrieben wurde. Somit können Sie die Makros `EVT_MENU` oder `EVT_TOOL` sowohl für Menüelemente als auch für die Toolbar-Buttons verwenden. Der Ereignis-Handle einer Toolbar erhält auch hier `wxCommandEvent` als Argument.

Handle einrichten	Beschreibung
`EVT_TOOL(id, func)`	Behandelt das `wxEVT_COMMAND_TOOL_CLICKED`-Ereignis (ein Synonym für `wxEVT_COMMAND_MENU_SELECTED`), das generiert wird, wenn ein Tool-Element mit der ID `id` angeklickt wird.
`EVT_MENU(id, func)`	wie `EVT_TOOL`
`EVT_TOOL_RANGE(id1, id2, func)`	Behandelt das `wxEVT_COMMAND_TOOL_CLICKED`-Ereignis (ein Synonym für `wxEVT_COMMAND_MENU_SELECTED`), das generiert wird, wenn ein Tool-Element zwischen dem Bereich `id1` und `id2` angeklickt wurde.
`EVT_MENU_RANGE(id1, id2, func)`	dito, wie `EVT_TOOL_RANGE`
`EVT_TOOL_RCLICKED(id, func)`	Behandelt das `wxEVT_COMMAND_TOOL_RCLICKED`-Ereignis, das generiert wird, wenn ein Tool-Element mit der ID `id` mit der rechten Maustaste angeklickt wird.
`EVT_TOOL_RCLICKED_RANGE(id1, id2, func)`	Behandelt das `wxEVT_COMMAND_TOOL_RCLICKED`-Ereignis, das generiert wird, wenn ein Tool-Element zwischen dem Bereich `id1` und `id2` mit der rechten Maustaste angeklickt wurde.
`EVT_TOOL_ENTER(id, func)`	Behandelt das `wxEVT_COMMAND_TOOL_RCLICKED`-Ereignis, das generiert wird, wenn der Mauszeiger in den Bereich des Tool-Elements kommt. Mit der eigentlichen `id` des Tool-Elements kommen Sie hier nicht mehr weiter, weil dies für andere Zwecke gebraucht wird. Daher sollten Sie für `id` beispielsweise `wxID_ANY` angeben und im Ereignis-Handle dann die Methode `wxCommandEvent::GetSelection` verwenden, um an die `id` des Toolbar-Elements zu kommen.

Tabelle 10.67 »wxToolBar«-Ereignisse behandeln

PopupMenu

Eine weitere Art, `wxMenu` zu verwenden, ist ein Kontextmenü, das mit `wxWindow::PopupMenu` erzeugt wird. Ein solches Kontextmenü erscheint in den meisten Anwendungen, wenn man die rechte Maustaste betätigt (siehe Abbildung 10.35).

Zum Einrichten eines Ereignis-Handles für ein solches Popup-Menü steht Ihnen das folgende Makro zur Verfügung:

```
EVT_CONTEXT_MENU(func)
```

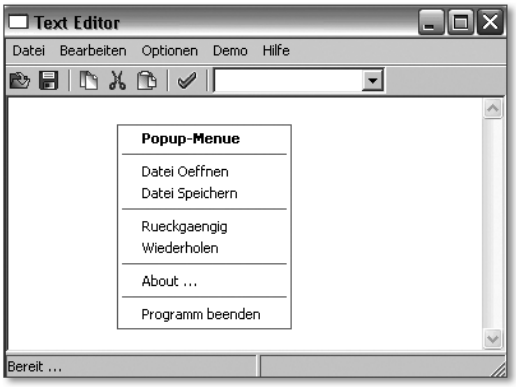

Abbildung 10.35 Ein Popup-Menü (Kontextmenü)

Sobald im (Client-)Fenster ein rechter Mausklick registriert wurde, wird der Ereignis-Handle aufgerufen. Dieser benötigt `wxContextMenuEvent` als Argument.

Im nächsten Schritt sollte man mit `wxContextMenuEvent::GetPosition` die Position des rechten Mausklicks ermitteln. Diese so ermittelte Position muss noch entsprechend auf die Anwendung (genauer den Client-Bereich) konvertiert werden, da sonst das Popup-Menü ein wenig nach rechts unten versetzt angezeigt würde. Diese Konvertierung geschieht mit der Methode `wxWindow::ScreenToClient`. Jetzt kann man wie gewöhnlich sein `wxMenu` zusammenstellen und mit dem Popup-Menü aufrufen. Die Methode `wxWindow::PopupMenu` hat folgende Syntax:

```
bool PopupMenu(
    wxMenu* menu,
    const wxPoint& pos = wxDefaultPosition ) ;

bool PopupMenu( wxMenu* menu, int x, int y );
```

Die Koordinaten `x` und `y` oder `pos` haben Sie zuvor mit `wxWindow::ScreenToClient` ermittelt und können Sie hierfür verwenden. Das Menü (`wxMenu`), das angezeigt werden soll, wird dem ersten Parameter übergeben. Somit sieht der komplette Vorgang, ein eigenes Kontextmenü (wie in Abbildung 10.35) zu erstellen, folgendermaßen aus:

```
// Ereignis-Handle einrichten
...
   EVT_CONTEXT_MENU(TextFrame::OnContextMenu)
...

// Ein Kontext-Menü - erscheint bei Rechtsklick auf
```

```
// die Maus im Textfeld
void TextFrame::OnContextMenu(wxContextMenuEvent& event) {
   // Position ermitteln
   wxPoint point = event.GetPosition();
   point = ScreenToClient(point);
   ShowContextMenu(point);
}

// Das Kontext-Menü
void TextFrame::ShowContextMenu(const wxPoint& pos) {
   wxMenu menu(wxT("Popup-Menue"));
   menu.Append(MENU_FILE_OPEN, wxT("Datei Öffnen"));
   menu.Append(MENU_FILE_SAVE, wxT("Datei Speichern"));
   menu.AppendSeparator();
   menu.Append(MENU_EDIT_UNDO, wxT("Rückgängig"));
   menu.Append(MENU_EDIT_REDO, wxT("Wiederholen"));
   menu.AppendSeparator();
   menu.Append(MENU_INFO_ABOUT, wxT("About ..."));
   menu.AppendSeparator();
   menu.Append(MENU_FILE_QUIT, wxT("Programm beenden"));
   PopupMenu(&menu, pos.x, pos.y);
}
```

10.3.13 Ein Beispiel – Text-Editor

Jetzt soll ein etwas umfangreicheres Beispiel zu `wxWidgets` gezeigt werden – ein einfacher und ausbaubarer Text-Editor. Hierbei wird alles bisher Beschriebene etwas ausführlicher dargestellt und in die Praxis umgesetzt. Vorgezogen wurden in diesem Beispiel einige Dialoge, die erst im nächsten Abschnitt (10.3.14, »Standarddialoge«) näher beschrieben werden. Der Text-Editor ist für sehr umfangreiche Texte nicht geeignet, nach 32 KB ist Schluss (es handelt sich hier nur um eine Demonstration).

[»] **Hinweis**

Wollen Sie nicht den gesamten Quelltext abtippen, finden Sie das Beispiel auch auf der Buch-CD.

[»] **Hinweis**

Zum Studieren eines guten Text-Editors, der mit `wxWidgets` geschrieben wurde und bei dem auch noch der Quellcode offenliegt, kann ich Ihnen Turbo Pad empfehlen (Webseite: *http://sourceforge.net/projects/turbopad*).

Hier also zunächst wieder die Header-Datei *base.h*:

```
// base.h
#ifndef _TEXT_H
#define _TEXT_H

#include "wx/wx.h"
#include "wx/fdrepdlg.h"
#include "wx/colordlg.h"
#include "wx/fontdlg.h"
#include "wx/numdlg.h"
#include "wx/textdlg.h"
#include "wx/fdrepdlg.h"
#include "wx/print.h"
#include "wx/printdlg.h"

class TextEditorApp : public wxApp {
   public:   virtual bool OnInit();
};

DECLARE_APP(TextEditorApp)

class TextFrame : public wxFrame {
public:
   TextFrame( const wxChar *title, int xpos, int ypos,
              int width, int height );
   ~TextFrame();

   void OnMenuFileOpen(wxCommandEvent &event);
   void OnMenuFileSave(wxCommandEvent &event);
   void OnMenuFileQuit(wxCommandEvent &event);
   void OnMenuInfoAbout(wxCommandEvent &event);
   void OnMenuOptionBackgroundColor(wxCommandEvent &event);
   void OnMenuOptionFont(wxCommandEvent &event);
   void OnMenuOptionDirectory(wxCommandEvent &event);
   void OnMenuEditGoto(wxCommandEvent &event);
   void OnMenuEditCopy(wxCommandEvent &event);
   void OnMenuEditCut(wxCommandEvent &event);
   void OnMenuEditPaste(wxCommandEvent &event);
   void OnMenuEditUndo(wxCommandEvent &event);
   void OnMenuEditRedo(wxCommandEvent &event);
   void OnMenuEditDelete(wxCommandEvent &event);
   void OnMenuEditSelect(wxCommandEvent &event);
   void OnMenuEditSearch(wxCommandEvent &event);
   void OnClose(wxCloseEvent &event);
   void OnText(wxCommandEvent &event);
   void ShowContextMenu(const wxPoint& pos);
```

```
    void OnContextMenu(wxContextMenuEvent& event);
    void OnMenuDemoChecked( wxCommandEvent &event );
    void OnMenuDemoRadio( wxCommandEvent& event);
    void OnCheckTool( wxCommandEvent& event);
protected:
    DECLARE_EVENT_TABLE()
private:
    wxTextCtrl *m_pTextCtrl;
    wxMenuBar *m_pMenuBar;
    wxMenu *m_pFileMenu;
    wxMenu *m_pEditMenu;
    wxMenu *m_pInfoMenu;
    wxMenu *m_pOptionMenu;
    wxMenu *m_pDemoMenu;
    wxFindReplaceDialog *m_dlgReplace;

    enum {
        MENU_FILE_OPEN,
        MENU_FILE_SAVE,
        MENU_FILE_QUIT,
        MENU_EDIT_UNDO,
        MENU_EDIT_REDO,
        MENU_EDIT_GOTO,
        MENU_EDIT_COPY,
        MENU_EDIT_CUT,
        MENU_EDIT_PASTE,
        MENU_EDIT_DELETE,
        MENU_EDIT_SELECT,
        MENU_EDIT_SEARCH,
        MENU_OPTION_BACKGROUND,
        MENU_OPTION_FONT,
        MENU_OPTION_DIRECTORY,
        MENU_DEMO_NORMAL,
        MENU_DEMO_CHECK1,
        MENU_DEMO_RADIO1,
        MENU_DEMO_RADIO2,
        MENU_DEMO_RADIO3,
        MENU_INFO_ABOUT,
        ID_TEXTCTRL
    };
};
#endif
```

Und jetzt der gesamte Quelltext von *base.cpp* (den Sie ebenfalls auf der Buch-CD finden):

```
#include "base.h"

IMPLEMENT_APP(TextEditorApp)

bool TextEditorApp::OnInit() {
   TextFrame *frame = new TextFrame(
      wxT("Text Editor"), 100, 100, 400, 300);
   frame->Show(TRUE);
   SetTopWindow(frame);
   return true;
}

TextFrame::TextFrame( const wxChar *title,
    int xpos, int ypos, int width, int height )
    : wxFrame( (wxFrame *) NULL, -1, title,
                wxPoint(xpos, ypos), wxSize(width, height))
{
  m_pTextCtrl = new wxTextCtrl(
     this, ID_TEXTCTRL, wxString(wxT("")),
     wxDefaultPosition, wxDefaultSize,
     wxTE_MULTILINE|wxTE_NOHIDESEL );
  // Eine neue Menübar
  m_pMenuBar = new wxMenuBar();
  // "Datei" - Menü
  m_pFileMenu = new wxMenu();

  ////////////////////////////////
  // Elemente zum Menü hinzufügen
  ////////////////////////////////

  // --- Möglichkeit 1: mit wxMenuItem und dann mit Append
  wxMenuItem *load = new wxMenuItem(
     m_pFileMenu, MENU_FILE_OPEN,
     wxT("&Öffnen\tCtrl-O"), wxT("Eine Datei öffnen ...") );
  wxBitmap bitmap_menu(wxT("open.bmp"), wxBITMAP_TYPE_BMP );
  load->SetBitmap(bitmap_menu);

  wxMenuItem *save = new wxMenuItem(
     m_pFileMenu, MENU_FILE_SAVE,
     wxT("&Speichern\tCtrl-S"), wxT("Datei speichern ..."));
  wxBitmap bitmap_menu2(
     wxT("save.bmp"), wxBITMAP_TYPE_BMP );
  save->SetBitmap(bitmap_menu2);
```

```
// Die einzelnen Elemente des "Datei"-Menüs hinzufügen
m_pFileMenu->Append(load);
m_pFileMenu->Append(save);
m_pFileMenu->AppendSeparator();

//-- Oder ab jetzt die 2. Möglichkeit, direkt mit Append --

m_pFileMenu->Append( MENU_FILE_QUIT,
                     wxT("&Beenden\tCtrl-Q"),
                     wxT("Beendet das Programm"));
// Die einzelnen Menü-Elemente jetzt zu "Datei" hinzufügen
m_pMenuBar->Append(m_pFileMenu, wxT("&Datei"));
// "Bearbeiten" - Menü
m_pEditMenu = new wxMenu();
m_pEditMenu->Append(MENU_EDIT_UNDO,
   wxT("Rückgängig\tCtrl-Z"), wxT("Rückgängig machen"));
m_pEditMenu->Append(MENU_EDIT_REDO,
   wxT("Wiederholen\tCtrl-W"), wxT("Wiederholen ..."));
m_pEditMenu->AppendSeparator();
m_pEditMenu->Append( MENU_EDIT_GOTO,
                     wxT("Gehe zu Zeile\tCtrl-J"),
                     wxT("Springt zu einer Zeile"));
m_pEditMenu->AppendSeparator();
m_pEditMenu->Append(MENU_EDIT_COPY,
   wxT("&Kopieren\tCtrl-C"), wxT("Auswahl kopieren"));
m_pEditMenu->Append( MENU_EDIT_CUT,
                     wxT("&Ausschneiden\tCtrl-X"),
                     wxT("Auswahl ausschneiden"));
m_pEditMenu->Append( MENU_EDIT_PASTE,
                     wxT("&Einfügen\tCtrl-V"),
                     wxT("Aus Zwischenablage einfügen"));
m_pEditMenu->AppendSeparator();

// Wir erzeugen ein weiteres Untermenü in "Bearbeiten"
wxMenu* subMenu = new wxMenu;
subMenu->Append( MENU_EDIT_DELETE,
                 wxT("&Löschen\tCtrl-D"),
                 wxT("Kompletten Text löschen"));
subMenu->Append( MENU_EDIT_SELECT,
                 wxT("&Markieren\tCtrl-A"),
                 wxT("Kompletten Text markieren") );
m_pEditMenu->Append(wxID_ANY, wxT("Alles ..."), subMenu);

m_pEditMenu->AppendSeparator();
m_pEditMenu->Append(MENU_EDIT_SEARCH,
   wxT("&Suchen\tCtrl-F"), wxT("Nach Text suchen"));
m_pMenuBar->Append(m_pEditMenu, wxT("&Bearbeiten"));
```

```
// Option Menü
m_pOptionMenu = new wxMenu();
m_pOptionMenu->Append(MENU_OPTION_BACKGROUND,
   wxT("&Hintergrund\tAlt-B"),
   wxT("Setzt die Hintergrundfarbe"));
m_pOptionMenu->Append( MENU_OPTION_FONT,
                       wxT("&Schriftarten\tAlt-F"),
                       wxT("Setzt die Schriftart") );
m_pOptionMenu->Append(MENU_OPTION_DIRECTORY,
                      wxT("&Verzeichnis\tAlt-D"),
                     wxT("Setzt das Arbeitsverzeichnis"));
m_pMenuBar->Append(m_pOptionMenu, wxT("&Optionen"));

// Demonstrationsmenü
m_pDemoMenu = new wxMenu();
m_pDemoMenu->Append(MENU_DEMO_NORMAL,
   wxT("Normal"), wxT("Ein normales Element"));
m_pDemoMenu->Enable(MENU_DEMO_NORMAL, false );
m_pDemoMenu->AppendSeparator();
m_pDemoMenu->AppendCheckItem(MENU_DEMO_CHECK1,
   wxT("Check1"), wxT("Ein Check-Element"));
m_pDemoMenu->Check(MENU_DEMO_CHECK1, true);
m_pDemoMenu->AppendSeparator();
m_pDemoMenu->AppendRadioItem(MENU_DEMO_RADIO1,
   wxT("Radio1"), wxT("Ein Radio-Element (1)"));
m_pDemoMenu->AppendRadioItem(MENU_DEMO_RADIO2,
   wxT("Radio2"), wxT("Ein Radio-Element (2)"));
m_pDemoMenu->AppendRadioItem(MENU_DEMO_RADIO3,
   wxT("Radio3"), wxT("Ein Radio-Element (3)"));

m_pMenuBar->Append(m_pDemoMenu, wxT("&Demo"));

// About menu
m_pInfoMenu = new wxMenu();
m_pInfoMenu->Append(MENU_INFO_ABOUT,
   wxT("&About"), wxT("Wer hat den Editor verbrochen?"));
m_pMenuBar->Append(m_pInfoMenu, wxT("&Hilfe"));

//////////////
// Die Toolbar
//////////////

wxToolBar* toolBar = new wxToolBar(this, wxID_ANY,
   wxDefaultPosition, wxDefaultSize,
   wxTB_HORIZONTAL|wxNO_BORDER|wxTB_FLAT);
```

```
  wxBitmap bitmap1(wxT("open.bmp"), wxBITMAP_TYPE_BMP );
  wxBitmap bitmap2(wxT("save.bmp"), wxBITMAP_TYPE_BMP );
  wxBitmap bitmap3(wxT("copy.bmp"), wxBITMAP_TYPE_BMP );
  wxBitmap bitmap4(wxT("cut.bmp"), wxBITMAP_TYPE_BMP );
  wxBitmap bitmap5(wxT("paste.bmp"), wxBITMAP_TYPE_BMP );
  wxBitmap bitmap6(wxT("check.bmp"), wxBITMAP_TYPE_BMP );

  toolBar->AddTool( MENU_FILE_OPEN, bitmap1, wxT("Öffnen"));
  toolBar->AddTool( MENU_FILE_SAVE, bitmap2,
                    wxT("Speichern"));
  toolBar->AddSeparator();
  toolBar->AddTool( MENU_EDIT_COPY, bitmap3,
                    wxT("Kopieren"));
  toolBar->AddTool( MENU_EDIT_CUT,  bitmap4,
                    wxT("Ausschneiden"));
  toolBar->AddTool( MENU_EDIT_PASTE, bitmap5,
                    wxT("Einfügen"));
  toolBar->AddSeparator();
  toolBar->AddTool(MENU_DEMO_CHECK1, bitmap6, bitmap6,
     wxT("Check-Button"), wxITEM_CHECK);
  toolBar->AddSeparator();
  // Zur Demonstration auch Combo-Box möglich
  wxComboBox* comboBox = new wxComboBox(toolBar, wxID_ANY);
  toolBar->AddControl(comboBox);

  toolBar->Realize();
  SetToolBar(toolBar);
  SetMenuBar(m_pMenuBar);
  // Statusbar
  CreateStatusBar(2);
  SetStatusText(wxT("Bereit ..."), 0);
}

TextFrame::~TextFrame() { }

// Tabelle der Ereignis-Handles
BEGIN_EVENT_TABLE(TextFrame, wxFrame)
   EVT_MENU(MENU_FILE_OPEN, TextFrame::OnMenuFileOpen)
   EVT_MENU(MENU_FILE_SAVE, TextFrame::OnMenuFileSave)
   EVT_MENU(MENU_FILE_QUIT, TextFrame::OnMenuFileQuit)
   EVT_MENU(MENU_EDIT_GOTO, TextFrame::OnMenuEditGoto)
   EVT_MENU(MENU_EDIT_COPY, TextFrame::OnMenuEditCopy)
   EVT_MENU(MENU_EDIT_CUT, TextFrame::OnMenuEditCut)
   EVT_MENU(MENU_EDIT_UNDO, TextFrame::OnMenuEditUndo)
   EVT_MENU(MENU_EDIT_REDO, TextFrame::OnMenuEditRedo)
   EVT_MENU(MENU_EDIT_PASTE, TextFrame::OnMenuEditPaste)
```

```
   EVT_MENU(MENU_EDIT_DELETE, TextFrame::OnMenuEditDelete)
   EVT_MENU(MENU_EDIT_SELECT, TextFrame::OnMenuEditSelect)
   EVT_MENU(MENU_EDIT_SEARCH, TextFrame::OnMenuEditSearch)
   EVT_MENU(MENU_INFO_ABOUT, TextFrame::OnMenuInfoAbout)
   EVT_MENU(MENU_OPTION_BACKGROUND,
      TextFrame::OnMenuOptionBackgroundColor)
   EVT_MENU(MENU_OPTION_FONT, TextFrame::OnMenuOptionFont)
   EVT_MENU(MENU_OPTION_DIRECTORY,
      TextFrame::OnMenuOptionDirectory)
   EVT_MENU(MENU_DEMO_CHECK1, TextFrame::OnMenuDemoChecked)
   EVT_MENU(MENU_DEMO_RADIO1, TextFrame::OnMenuDemoRadio)
   EVT_MENU(MENU_DEMO_RADIO2, TextFrame::OnMenuDemoRadio)
   EVT_MENU(MENU_DEMO_RADIO3, TextFrame::OnMenuDemoRadio)
   EVT_TOOL_ENTER(wxID_ANY, TextFrame::OnCheckTool)
   EVT_CONTEXT_MENU(TextFrame::OnContextMenu)
   EVT_CLOSE(TextFrame::OnClose)
   EVT_TEXT(ID_TEXTCTRL, TextFrame::OnText)
END_EVENT_TABLE()

// Datei->Öffnen
void TextFrame::OnMenuFileOpen(wxCommandEvent &event) {
  wxFileDialog *dlg = new wxFileDialog(
     this, wxT("Öffnen einer Text-Datei"), wxT(""),wxT(""),
     wxT("Alle Dateien(*.*)|*.*|Text Dateien(*.txt)|*.txt"),
     wxOPEN, wxDefaultPosition);
   if ( dlg->ShowModal() == wxID_OK )   {
      m_pTextCtrl->LoadFile(dlg->GetFilename());
      SetStatusText(dlg->GetFilename(), 0);
      wxString s = wxT("Text Editor - ");
      s.Append(dlg->GetFilename());
      SetTitle( s );
   }
   dlg->Destroy();
}

// Datei->Speichern
void TextFrame::OnMenuFileSave(wxCommandEvent &event) {
   wxFileDialog *dlg = new wxFileDialog(
     this, wxT("Speichern einer Text-Datei"),
     wxT(""),wxT(""),
     wxT("Alle Dateien(*.*)|*.*|Text Dateien(*.txt)|*.txt"),
     wxSAVE, wxDefaultPosition );
   if ( dlg->ShowModal() == wxID_OK ) {
      m_pTextCtrl->SaveFile(dlg->GetPath());
      SetStatusText(dlg->GetFilename(), 0);
      wxString s = wxT("Text Editor - ");
```

```
      s.Append(dlg->GetFilename());
      SetTitle( s );
   }
   dlg->Destroy();
   m_pTextCtrl->DiscardEdits();
}

// Datei->Beenden
void TextFrame::OnMenuFileQuit(wxCommandEvent &event) {
   Close(FALSE);
}

// Hilfe->About
void TextFrame::OnMenuInfoAbout(wxCommandEvent &event) {
   wxMessageDialog *dlg = new wxMessageDialog(
      this, wxT("(c) 2006 P.R.O.N.I.X\nJ.Wolf & Co.\n"),
      wxT("About..."), wxOK | wxICON_INFORMATION );
   dlg->ShowModal();
}

// Bearbeiten->Gehe zu Zeile
void TextFrame::OnMenuEditGoto(wxCommandEvent& event) {
   wxString s;
   s += wxString::Format(
      wxT("Eine Zeile zwischen 1 und %d auswaehlen"),
      m_pTextCtrl->GetNumberOfLines() );

   wxNumberEntryDialog dialog(
      this, s, wxT("Nummer eingeben: "),
      wxT("Gehe zu Zeile"),
      1, 1, m_pTextCtrl->GetNumberOfLines() );

   if (dialog.ShowModal() == wxID_OK) {
      long value = dialog.GetValue();
      m_pTextCtrl->SetInsertionPoint(
         m_pTextCtrl->XYToPosition(0, value-1) );
   }
}

// Bearbeiten->Kopieren
void TextFrame::OnMenuEditCopy(wxCommandEvent& event) {
   if( m_pTextCtrl->CanCopy() ) {
      m_pTextCtrl->Copy();
   }
   else {
      SetStatusText(
```

```
            wxT("Nichts zum Kopieren vorhanden ...?!"));
      }
   }

   // Bearbeiten->Ausschneiden
   void TextFrame::OnMenuEditCut(wxCommandEvent& event) {
      if( m_pTextCtrl->CanCut() ) {
         m_pTextCtrl->Cut();
      }
      else {
         SetStatusText(
            wxT("Nichts zum Ausschneiden vorhanden ...?!"));
      }
   }

   // Bearbeiten->Einfügen
   void TextFrame::OnMenuEditPaste(wxCommandEvent& event) {
      if( m_pTextCtrl->CanPaste() ) {
         m_pTextCtrl->Paste();
      }
      else {
         SetStatusText(
            wxT("Nichts zum Einfügen vorhanden ...?!"));
      }
   }

   // Bearbeiten->Rückgängig
   void TextFrame::OnMenuEditUndo(wxCommandEvent& event) {
      if( m_pTextCtrl->CanUndo()) {
         m_pTextCtrl->Undo();
      }
      else {
         SetStatusText(
            wxT("Nichts zum Rückgängigmachen vorhanden ?!"));
      }
   }

   // Bearbeiten->Wiederholen
   void TextFrame::OnMenuEditRedo(wxCommandEvent& event) {
      if( m_pTextCtrl->CanRedo() ) {
         m_pTextCtrl->Redo();
      }
      else {
         SetStatusText(
            wxT("Nichts zum Wiederholen vorhanden ...?!"));
      }
```

```
}

// Bearbeiten->Alles...->Löschen
void TextFrame::OnMenuEditDelete(wxCommandEvent& event) {
   m_pTextCtrl->Clear();
   m_pTextCtrl->MarkDirty();
}

// Bearbeiten->Alles...->Auswählen
void TextFrame::OnMenuEditSelect(wxCommandEvent& event) {
   m_pTextCtrl->SetSelection(-1, -1);
}

// Bearbeiten->Suchen
void TextFrame::OnMenuEditSearch(wxCommandEvent& event) {
   wxTextEntryDialog dialog(
      this, wxT("Wonach wollen Sie suchen?"),
      wxT("Bitte einen String eingeben"), wxT(""),
      wxOK | wxCANCEL );
   if (dialog.ShowModal() == wxID_OK) {
      // Such-String einlesen
      wxString s = dialog.GetValue();
      for( int line =1;
           line < m_pTextCtrl->GetNumberOfLines();
           ++line )
      {
         bool search = true;
         // Komplette Zeile einlesen ...
         wxString tmp = m_pTextCtrl->GetLineText(line);
         // Anfangsposition  des Suchstrings ermitteln
         int found = tmp.Find( s );
         if (found != -1) {
            int pos = m_pTextCtrl->XYToPosition(
               found,  line );
            m_pTextCtrl->SetSelection( pos, pos + s.Len());
            wxMessageDialog dialog(
               NULL,wxT("Wollen Sie weitersuchen?"),
               wxT("Weitersuchen...?"),
               wxNO_DEFAULT|wxYES_NO|wxICON_QUESTION);
            switch ( dialog.ShowModal() ) {
               case wxID_YES: break;
               case wxID_NO:
                  search = false;
                  break;
               default: break;
            }
```

```
            if( !search )
               break;
         }
      }
   }
}

// Option->Hintergrund
void TextFrame::OnMenuOptionBackgroundColor(
   wxCommandEvent &event)
{
   wxColourData colourData;
   wxColour colour = m_pTextCtrl->GetBackgroundColour();
   colourData.SetColour(colour);
   colourData.SetChooseFull(true);
   wxColourDialog *dlg = new wxColourDialog(
      this, &colourData );
   if ( dlg->ShowModal() == wxID_OK ) {
      colourData = dlg->GetColourData();
      m_pTextCtrl->SetBackgroundColour(
         colourData.GetColour() );
      m_pTextCtrl->Refresh();
   }
   dlg->Destroy();
}

// Optionen->Schriftarten
void TextFrame::OnMenuOptionFont(wxCommandEvent& event) {
   wxFontData fontData;
   wxFont font;
   wxColour colour;

   font = m_pTextCtrl->GetFont();
   fontData.SetInitialFont(font);
   colour = m_pTextCtrl->GetForegroundColour();
   fontData.SetColour(colour);
   fontData.SetShowHelp(true);

   wxFontDialog *dlg = new wxFontDialog(this, &fontData);
   if ( dlg->ShowModal() == wxID_OK ) {
      fontData = dlg->GetFontData();
      font = fontData.GetChosenFont();
      m_pTextCtrl->SetFont(font);
      m_pTextCtrl->SetForegroundColour(
         fontData.GetColour() );
      m_pTextCtrl->Refresh();
```

```
   }
   dlg->Destroy();
}

// Optionen->Verzeichnis
void TextFrame::OnMenuOptionDirectory(wxCommandEvent& event)
{
   wxDirDialog *dlg = new wxDirDialog(
      this, wxT("Ein neues Arbeitsverzeichnis auswählen"),
      wxGetCwd() );
   if ( dlg->ShowModal() == wxID_OK ) {
      wxSetWorkingDirectory(dlg->GetPath());
   }
   dlg->Destroy();
}

// Anwendung wird beendet - überprüfen, ob die Textdatei
// verändert wurde und noch nicht abgespeichert ist
void TextFrame::OnClose(wxCloseEvent& event) {
   bool destroy = true;
   if ( event.CanVeto() ) {
      if ( m_pTextCtrl->IsModified() ){
         wxMessageDialog *dlg = new wxMessageDialog(
            this, wxT("Der Text wurde geändert!\n")
            wxT("Wollen Sie wirklich beenden?"),
            wxT("Text geändert!"),
            wxYES_NO | wxNO_DEFAULT | wxICON_QUESTION );
         int result = dlg->ShowModal();
         if ( result == wxID_NO ) {
            event.Veto();
            destroy = false;
         }
      }
   }
   if ( destroy ) {
      Destroy();
   }
}
// Informationen-Textfeld für die Statuszeile
void TextFrame::OnText( wxCommandEvent &event ) {
   long x, y;
   m_pTextCtrl->PositionToXY(
      m_pTextCtrl->GetInsertionPoint(), &x, &y );
   wxString s;
   s += wxString::Format(
      wxT("Insg. %d Zeilen: %d Zeichen (%d: %d)"),
```

```
      m_pTextCtrl->GetNumberOfLines(),
      m_pTextCtrl->GetLastPosition(), y+1, x+1 );
   SetStatusText( s, 1 );
}

// Ein Kontext-Menü - erscheint bei Rechtsklick
// auf die Maus im Textfeld
void TextFrame::OnContextMenu(wxContextMenuEvent& event) {
   // Position ermitteln
   wxPoint point = event.GetPosition();
   point = ScreenToClient(point);
   ShowContextMenu(point);
}

// Das Kontext-Menü
void TextFrame::ShowContextMenu(const wxPoint& pos) {
   wxMenu menu(wxT("Popup-Menue"));
   menu.Append(MENU_FILE_OPEN, wxT("Datei öffnen"));
   menu.Append(MENU_FILE_SAVE, wxT("Datei speichern"));
   menu.AppendSeparator();
   menu.Append(MENU_EDIT_UNDO, wxT("Rückgängig"));
   menu.Append(MENU_EDIT_REDO, wxT("Wiederholen"));
   menu.AppendSeparator();
   menu.Append(MENU_INFO_ABOUT, wxT("About ..."));
   menu.AppendSeparator();
   menu.Append(MENU_FILE_QUIT, wxT("Programm beenden"));
   PopupMenu(&menu, pos.x, pos.y);
}

// Checkfelder im Menü "Demo" auswerten
void TextFrame::OnMenuDemoChecked( wxCommandEvent &event ) {
   if( event.IsChecked()) {
      wxMessageDialog *dlg = new wxMessageDialog(
         this, wxT("Check-Menu wurde aktiviert"),
         wxT("Hinweis..."), wxOK | wxICON_INFORMATION );
      dlg->ShowModal();
   }
   else {
      wxMessageDialog *dlg = new wxMessageDialog(
         this, wxT("Check-Menu wurde DE-aktiviert"),
         wxT("Hinweis..."), wxOK | wxICON_INFORMATION );
      dlg->ShowModal();
   }
}
```

```
// Radio-Button im Menü "Demo" auswerten
void TextFrame::OnMenuDemoRadio( wxCommandEvent& event) {
    int id = event.GetId();
    wxString s;
    switch ( id ) {
       case MENU_DEMO_RADIO1:
          s.Append(wxT("Radio1 ist aktiv"));
          break;
       case MENU_DEMO_RADIO2:
          s.Append(wxT("Radio2 ist aktiv"));
          break;
       case MENU_DEMO_RADIO3:
          s.Append(wxT("Radio3 ist aktiv"));
          break;
       default: break;
    }
    wxMessageDialog *dlg =
       new wxMessageDialog(
          this, s, wxT("Hinweis..."),
          wxOK | wxICON_INFORMATION );
      dlg->ShowModal();
}

// Demonstriert im Grunde nur das Makro EVT_TOOL_ENTER
void TextFrame::OnCheckTool( wxCommandEvent& event) {
   int index = event.GetSelection();
   if( index != -1 ) {
      switch (index) {
         case MENU_FILE_OPEN:
            SetStatusText(
              wxT("EVT_TOOL_ENTER: Eine Datei öffnen"), 0);
            break;
         case MENU_FILE_SAVE:
            SetStatusText(
              wxT("EVT_TOOL_ENTER: Eine Datei speichern"), 0);
            break;

         // ... usw.
         case MENU_DEMO_CHECK1:
            SetStatusText(
               wxT("EVT_TOOL_ENTER: Check-Tool"), 0);
            break;
         default: break;
      }
   }
}
```

Das Programm bei der Ausführung:

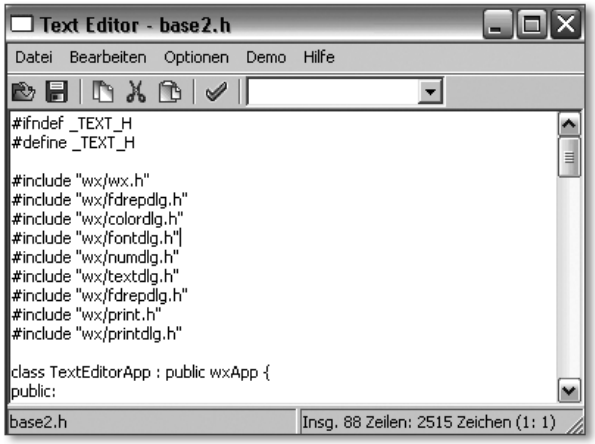

Abbildung 10.36 Der Text-Editor im Einsatz

10.3.14 Standarddialoge

Im letzten Beispiel haben Sie bei den Optionen und der Suche nach einem Text bzw. beim Springen zu einer Zeile einige Standarddialoge verwendet, die von `wxWidgets` angeboten werden. In diesem Abschnitt soll darauf eingegangen werden. Wenn Sie weitere Details erfahren möchten, sollten Sie die Referenz zur Hand nehmen.

Dialoge zur Anzeige von Informationen

Zur Anzeige von Informationen bietet Ihnen `wxWidgets` verschiedene Möglichkeiten. Einige der gebräuchlichsten sind `wxMessageDialog`, `wxProgressDialog`, `wxBusyInfo` und `wxShowTip`. Auf den Dialog `wxMessageDialog` wurde ja bereits näher eingegangen, weshalb hier auf eine Wiederholung verzichtet wird.

wxProgressDialog

`wxProgressDialog` ist eine Art Fortschrittsanzeige, die darüber informiert, wie lange ein Anwender noch warten muss, bis ein bestimmter Vorgang fertig abgearbeitet ist (siehe Abbildung 10.37).

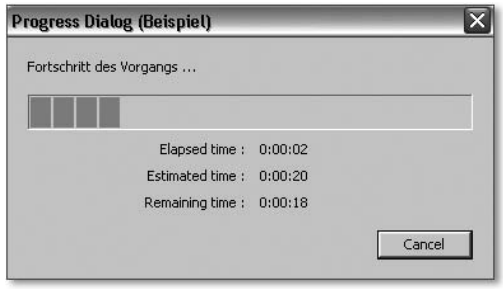

Abbildung 10.37 wxProgressDialog

Die Syntax des Konstruktors sieht folgendermaßen aus:

```
wxProgressDialog(
   const wxString& title,
   const wxString& message,
   int maximum = 100,
   wxWindow * parent = NULL,
   int style = wxPD_AUTO_HIDE | wxPD_APP_MODAL );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `title` – der Dialogtitel, der in der Titelleiste angezeigt wird
- `message` – die Nachricht, die über der Zustandsleiste angezeigt wird
- `maximum` – der maximale Wert für die Zustandsleiste
- `parent` – das Eltern-Fenster
- `style` – der Stil von `wxProgressDialog` (siehe Tabelle 10.68)

Style (wxProgressDialog)	Beschreibung
`wxPD_APP_MODAL`	Macht den Zustandsdialog modal. Wenn Sie diesen Stil nicht einsetzen, kann das Eltern-Fenster (`parent`) weiterhin verwendet werden.
`wxPD_AUTO_HIDE`	Lässt den Zustandsdialog sofort verschwinden (zerstört diesen aber nicht!), wenn der maximale Wert (`maximum`) erreicht wurde.
`wxPD_CAN_ABORT`	Fügt einen Cancel-Button zum Dialog hinzu, mit dem Sie den Dialog vorzeitig abbrechen können, bevor `maximum` erreicht wird.
`wxPD_ELAPSED_TIME`	Hiermit wird die bereits verstrichene Zeit des Zustandsdialogs angezeigt.
`wxPD_ESTIMATED_TIME`	Hiermit wird die Zeit angegeben, die voraussichtlich benötigt wird, bis der Vorgang beendet ist.

Tabelle 10.68 Die Stile von »wxProgressDialog«

Style (wxProgressDialog)	Beschreibung
`wxPD_REMAINING_TIME`	Hiermit wird die noch verbleibende Zeit (Countdown) angezeigt.

Tabelle 10.68 Die Stile von »wxProgressDialog« (Forts.)

Außerdem sind in `wxProgressDialog` zwei Methoden definiert:

Methode	Beschreibung
`void Resume();`	Wenn ein Zustandsdialog vom Anwender mit Cancel abgebrochen wurde, können Sie ihn mit dieser Methode fortsetzen.

`virtual bool Update(`
`  int value,`
`  const wxString& newmsg = "",`
`  bool *skip = NULL );`
	Erneuert den Dialog und setzt den neuen Wert und gegebenenfalls eine neue Nachricht. Mit `value` geben Sie den neuen Wert für den Progresszustand an. Dieser Wert muss kleiner sein als der Wert `maximum`. Mit `newmsg` können Sie eine neue Nachricht über den Zustandsbalken einfügen. Wenn der Skip-Button seit dem letzten Update betätigt wurde, ist der Wert `skip true`.

Tabelle 10.69 Die Methoden von »wxProgressDialog«

wxBusyInfo

Für die Anzeige eines Zustands können Sie auch die Klasse `wxBusyInfo` (siehe Abbildung 10.38) verwenden. Diese wird gewöhnlich eingesetzt, um dem Anwender anzuzeigen, dass er etwas warten muss.

Abbildung 10.38 wxBusyInfo

Bei dieser Klasse wird nur ein Konstruktor benötigt:

```
wxBusyInfo(const wxString& msg, wxWindow* parent = NULL);
```

Damit erzeugen Sie eine Busy-Wait-Information für den Anwender. Mit `msg` geben Sie die Nachricht an, die darin angezeigt werden soll. `parent` ist hierbei das Eltern-Fenster. Beachten Sie außerdem, dass das Eltern-Fenster nicht geschlossen wird, während die Busy-Info angezeigt wird, wenn Sie für `parent` nicht `NULL` verwenden.

Bevor Sie `wxBusyInfo` einsetzen, sollten Sie `wxWindowDisabler` verwenden, um alle derzeit geöffneten Fenster der Anwendung zu deaktivieren.

wxShowTip

Viele Anwendungen zeigen bei ihrem Start einen Tipp an. Das ist recht nützlich, weil man dadurch über die Software einiges lernen kann. `wxWidgets` bietet hierfür `wxShowTip` an:

Abbildung 10.39 wxShowTip

Die Funktion, mit der `wxShowTip` erzeugt wird, sieht wie folgt aus:

```
bool wxShowTip(
      wxWindow *parent,
      wxTipProvider *tipProvider,
      bool showAtStartup = true );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster für den modalen Dialog
- `tipProvider` – ein Objekt, das den Text für den Tipp von einer Datei holt. Dieses Objekt wird mit `wxCreateFileTipProvider` erzeugt.
- `ShowAtStartup` – wenn hierfür `true` verwendet wird, werden die Tipps gleich beim Starten der Anwendung angezeigt. Soll dies nicht geschehen, muss hier `false` eingesetzt werden. Dies ist die Initialisierungsvariable für die Checkbox, die den Tipp-Dialog anzeigt.

Die Tipps werden in einer gewöhnlichen Textdatei geschrieben. Die einzelnen Tipps werden durch ein Newline-Zeichen getrennt. `wxCreateFileTipProvider` verwendet also für jeden Tipp eine neue Zeile.

All together (ein Beispiel)

Im folgenden Beispiel sehen Sie nochmals alle drei hier erwähnten Dialoge (`wxProgressDialog`, `wxBusyInfo` und `wxShowTip`) in der Praxis.

Zunächst die Header-Datei *base.h*:

```
// base.h
#ifndef BASIC_H
#define BASIC_H

class wxFrameDemoApp : public wxApp {
   public: virtual bool OnInit();
};

class BasicFrame : public wxFrame {
private:
   wxMenuBar *MenuBar;
   wxMenu    *ExampleMenu;
   enum {
     MENU_DIALOG_PROGRESS,
     MENU_DIALOG_BUSY
   };
   // Ereignis-Tabelle einrichten
   DECLARE_EVENT_TABLE()
public:
   BasicFrame( const wxChar *title,
               int xpos, int ypos,
               int width, int height);
   ~BasicFrame();
   // Methoden, die auf Ereignisse reagieren
   void OnMenuDialogProgress (wxCommandEvent &event);
   void OnMenuDialogBusy(wxCommandEvent &event);
};
#endif
```

Jetzt noch die Quelldatei *base.cpp*:

```
// base.cpp
#include <wx/wx.h>
#include "base.h"
#include "wx/progdlg.h"
#include "wx/busyinfo.h"
#include "wx/tipdlg.h"

IMPLEMENT_APP(wxFrameDemoApp)

bool wxFrameDemoApp::OnInit() {
   BasicFrame *frame = new BasicFrame(
      wxT("Demonstriert einige Dialoge"),
      50, 50, 300, 200 );
```

```
   frame->Show(TRUE);
   SetTopWindow(frame);
   return TRUE;
}

BasicFrame::BasicFrame (
   const wxChar *title,
   int xpos, int ypos,
   int width, int height)
   : wxFrame ( (wxFrame *) NULL,
               -1, title,
               wxPoint(xpos, ypos),
               wxSize(width, height), wxDEFAULT_FRAME_STYLE)
{
   // Eine Menübar erzeugen
   MenuBar = new wxMenuBar();
   // Ein Menü erzeugen
   ExampleMenu = new wxMenu();
   ExampleMenu->Append(
      MENU_DIALOG_PROGRESS, wxT("&wxProgressDialog") );
   ExampleMenu->Append(
      MENU_DIALOG_BUSY, wxT("&wxBusyInfo") );
   MenuBar->Append( ExampleMenu, wxT("&Dialoge"));
   SetMenuBar(MenuBar);

   // Einen Tipp zum Programmstart anzeigen ...
   static size_t s_index = (size_t)-1;
   if ( s_index == (size_t)-1 ) {
      srand(time(NULL));

      s_index = rand() % 5;
   }
   wxTipProvider *tipProvider = wxCreateFileTipProvider(
      wxT("tips.txt"), s_index);
   wxShowTip(this, tipProvider, true);
   delete tipProvider;
   // Eine einfache Statusbar mit zwei Spalten erstellen
   CreateStatusBar(2);
}

BasicFrame::~BasicFrame() { }
// Ereignis-Tabelle
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   EVT_MENU(MENU_DIALOG_PROGRESS,
      BasicFrame::OnMenuDialogProgress)
   EVT_MENU(MENU_DIALOG_BUSY,
      BasicFrame::OnMenuDialogBusy)
```

```
END_EVENT_TABLE()

void BasicFrame::OnMenuDialogProgress(wxCommandEvent &event)
{
   static const int max = 10;
   wxProgressDialog dialog(
      wxT("Progress Dialog (Beispiel)"),
      wxT("Fortschritt des Vorgangs ..."), max, this,
      wxPD_CAN_ABORT | wxPD_APP_MODAL | wxPD_ELAPSED_TIME |
      wxPD_ESTIMATED_TIME | wxPD_REMAINING_TIME );

   bool cont = true;
   for ( int i = 0; i <= max; i++ ) {
      wxSleep(1);
      if ( i == max )
         cont = dialog.Update(
            i, wxT("Vorgang erfolgreich abgeschlossen"));
       else
          cont = dialog.Update(i);
       // Wurde der Vorgang vorzeitig abgebrochen?
       if ( !cont ) {
          if( wxMessageBox (
             wxT("Wollen Sie wirklich beenden?"),
             wxT("Vorgang beenden?"),
             wxYES_NO | wxICON_QUESTION) == wxYES )
                break;
          dialog.Resume();
       }
   }
   if ( !cont )
      SetStatusText(
         wxT("Vorgang wurde vorzeitig abgebrochen"), 0);
   else
      SetStatusText(
         wxT("Vorgang wurde ordentlich beendet"), 0);
}

void BasicFrame::OnMenuDialogBusy(wxCommandEvent &event) {

   // Alle Fenster der Anwendung abschalten ...
   wxWindowDisabler disableAll;
   wxBusyInfo info(wxT("Berechnung in Arbeit ..."), this);
   for (int i = 0; i < 5; i++) {
       // Berechnung_ausführen()
       wxSleep(1);
   }
   SetStatusText(
```

```
        wxT("Berechnung erfolgreich durchgeführt"), 0);
}
```

Dialoge für Dateien und Verzeichnisse

Für die Datei- und Verzeichnisdialoge können Sie – wie bereits im Listing des Text-Editors – die beiden Klassen `wxFileDialog` und `wxDirDialog` verwenden.

wxFileDialog

Die Klasse `wxFileDialog` wird verwendet, um eine oder mehrere Dateien zu öffnen oder zu speichern:

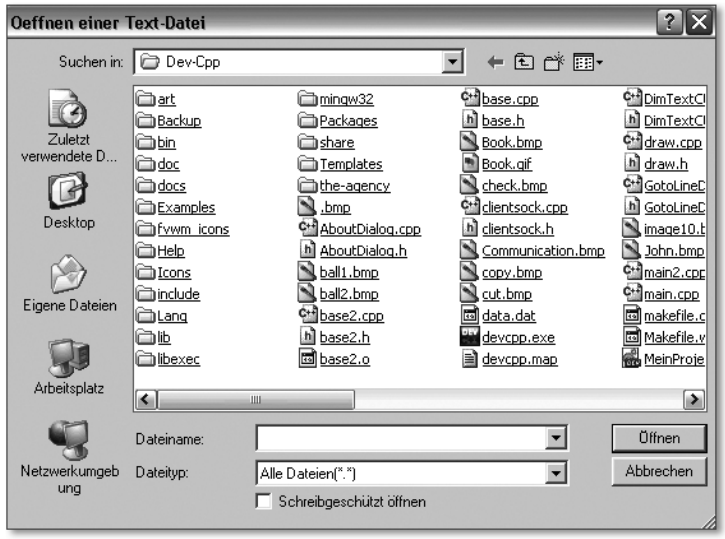

Abbildung 10.40 wxFileDialog

Im Grunde kann man fast alles gleichzeitig mit dem Konstruktor von `wxFileDialog` einrichten, aber es ist auch möglich, über die vielen `Set`- und `Get`-Zugriffsmethoden auf die einzelnen Elemente von `wxFileDialog` zuzugreifen. Wegen des Umfangs begnügen wir uns hier mit dem Nötigsten. Wenn Sie weitere Informationen benötigen, sollten Sie sich die Referenz ansehen.

Die Syntax des Konstruktors der Klasse `wxFileDialog` sieht folgendermaßen aus:

```
wxFileDialog(
   wxWindow* parent,
   const wxString& message = "Choose a file",
   const wxString& defaultDir = "",
   const wxString& defaultFile = "",
   const wxString& wildcard = "*.*",
```

```
    long style = 0,
    const wxPoint& pos = wxDefaultPosition );
```

Um den Dialog immer im Vordergrund zu halten, sollten Sie anschließend die Methode `wxFileDialog::showModal` verwenden; damit können Sie den Dialog modal anzeigen.

Die einzelnen Parameter des Konstruktors haben folgende Bedeutung:

- `parent` – das Eltern-Fenster
- `message` – die Nachricht, die in der Titelleiste des Dialogs angezeigt wird
- `defaultDir` – das Standardverzeichnis, das der Dialog auflisten soll, oder ein leerer String
- `defaultFile` – die Standarddatei, die der Dialog zum Öffnen bzw. Speichern verwenden soll, oder ein leerer String
- `wildcard` – eine gewöhnliche Wildcard, die angibt, welche Dateitypen geöffnet werden sollen, beispielsweise: `"Alle Dateien(*.*)|*.*|Text Dateien (*.txt)|*.txt"`
- `style` – der Stil von `wxFileDialog` (siehe Tabelle 10.70)
- `pos` – normalerweise die Position des Dialogfensters. Dieser Parameter ist allerdings nicht implementiert.

style (wxFileDialog)	Beschreibung
`wxOPEN`	der Dialog zum Öffnen einer Datei
`wxSAVE`	der Dialog zum Speichern einer Datei
`wxOVERWRITE_PROMPT`	Nur für einen Dialog zum Speichern gedacht. Damit wird ein Prompt angezeigt, mit dem der Anwender bestätigen muss, ob er eine bereits vorhandene Datei überschreiben will.
`wxHIDE_READONLY`	Zeigt nicht die Checkbox Schreibgeschützt öffnen, womit Sie eine Datei Read-only anzeigen.
`wxFILE_MUST_EXIST`	Der Anwender kann nur Dateien auswählen, die gerade aktuell existieren.
`wxMULTIPLE`	Damit können mehrere Dateien auf einmal ausgewählt und geöffnet werden.
`wxCHANGE_DIR`	Damit kann eine Datei auch außerhalb des aktuellen Arbeitsverzeichnisses vom Anwender ausgewählt bzw. gespeichert werden.

Tabelle 10.70 Die Stile von »wxFileDialog«

Wenn Sie den Konstruktor von `wxFileDialog` aufgerufen haben, sollten Sie anschließend die Methode `ShowModal` auf `wxID_OK` testen und sehen, was zurückgegeben wird, wenn der Anwender die Auswahl bestätigt.

Wie bereits erwähnt, besitzt dieser Dialog viele Zugriffsmethoden. In der Praxis kommt man aber gewöhnlich mit dem Ermitteln des Dateinamens oder gelegentlich auch des Verzeichnisses zurecht:

Methode	Beschreibung
`wxString GetDirectory() const;`	Gibt das Standardverzeichnis zurück.
`wxString GetFilename() const;`	Gibt den Dateinamen zurück, der ausgewählt wurde.
`void GetFilenames( wxArrayString& filenames) const;`	Füllt ein Array mit Dateinamen, die vom Anwender ausgewählt wurden. Diese Methode sollte nur in Verbindung mit dem Flag `wxMULTIPLE` eingesetzt werden, ansonsten verwendet man `GetFilename`.

Tabelle 10.71 Wichtige Methoden von »wxFileDialog«

In unserem Beispiel zum Text-Editor in Abschnitt 10.3.13, »Ein Beispiel – Text-Editor«, wurde bereits der Dialog zum Speichern und Öffnen einer Datei verwendet. Der Codeausschnitt hierzu sah folgendermaßen aus:

```
// Datei->Öffnen
void TextFrame::OnMenuFileOpen(wxCommandEvent &event) {
  wxFileDialog *dlg = new wxFileDialog(
     this, wxT("Öffnen einer Text-Datei"), wxT(""),wxT(""),
     wxT("Alle Dateien(*.*)|*.*|Text Dateien(*.txt)|*.txt"),
     wxOPEN, wxDefaultPosition);
   if ( dlg->ShowModal() == wxID_OK )   {
      m_pTextCtrl->LoadFile(dlg->GetFilename());
      SetStatusText(dlg->GetFilename(), 0);
      wxString s = wxT("Text Editor - ");
      s.Append(dlg->GetFilename());
      SetTitle( s );
   }
   dlg->Destroy();
}

// Datei->Speichern
void TextFrame::OnMenuFileSave(wxCommandEvent &event) {
   wxFileDialog *dlg = new wxFileDialog(
     this, wxT("Speichern einer Text-Datei"),
     wxT(""),wxT(""),
     wxT("Alle Dateien(*.*)|*.*|Text Dateien(*.txt)|*.txt"),
     wxSAVE, wxDefaultPosition );
   if ( dlg->ShowModal() == wxID_OK ) {
      m_pTextCtrl->SaveFile(dlg->GetPath());
      SetStatusText(dlg->GetFilename(), 0);
```

```
        wxString s = wxT("Text Editor - ");
        s.Append(dlg->GetFilename());
        SetTitle( s );
    }
    dlg->Destroy();
    m_pTextCtrl->DiscardEdits();
}
```

wxDirDialog

`wxDirDialog` erlaubt dem Anwender, sich sein lokales oder ein Netzwerkverzeichnis auszuwählen (siehe Abbildung 10.41). Natürlich kann dabei auch gleich ein neues Verzeichnis erzeugt werden.

Abbildung 10.41 wxDirDialog

Der Konstruktor von `wxDirDialog` sieht folgendermaßen aus:

```
wxDirDialog(
    wxWindow* parent,
    const wxString& message = "Choose a directory",
    const wxString& defaultPath = "",
    long style = wxDD_DEFAULT_STYLE,
    const wxPoint& pos = wxDefaultPosition,
    const wxSize& size = wxDefaultSize,
    const wxString& name = "wxDirCtrl" );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster
- `message` – eine Nachricht, die im Dialog angezeigt wird, nicht auf der Titelleiste
- `defaultPath` – der Vorgabe-Pfad oder ein leerer String

- `style` – der Stil von `wxDirDialog` (siehe Tabelle 10.72)
- `pos` – die Position (wird unter MS Windows ignoriert)
- `size` – die Größe (wird unter MS Windows ignoriert)
- `name` – wird nicht verwendet.

style (wxDirDialog)	Beschreibung
`wxDD_NEW_DIR_BUTTON`	Fügt den Button zum Erzeugen eines neuen Verzeichnisses hinzu.

Tabelle 10.72 Die Stile von »wxDirDialog«

Nach dem Erzeugen von `wxDirDialog` sollten Sie die Methode `ShowModal` auf den Wert von `wxID_OK` testen, der zurückgegeben wird, wenn der Anwender eine Verzeichnisauswahl bestätigt.

Auch hier sind wieder einige Methoden implementiert, von denen Sie allerdings gewöhnlich nur die Methode `GetPath` benötigen, die einen String des vorgegebenen oder vom Anwender ausgewählten Pfads zurückgibt. Im Beispiel des Text-Editors haben Sie für `wxDirDialog` folgenden Codeausschnitt gesehen:

```
// Optionen->Verzeichnis
void TextFrame::OnMenuOptionDirectory(wxCommandEvent& event)
{
   wxDirDialog *dlg = new wxDirDialog(
      this, wxT("Ein neues Arbeitsverzeichnis auswählen"),
      wxGetCwd() );
   if ( dlg->ShowModal() == wxID_OK ) {
      wxSetWorkingDirectory(dlg->GetPath());
   }
   dlg->Destroy();
}
```

Dialoge zum Auswählen

Zum Auswählen von Farben oder Schriftarten stehen Ihnen die Klassen `wxColourDialog` und `wxFontDialog` zur Verfügung. Auch für die Auswahl einzelner oder mehrerer Text-Strings stehen mit `wxSingleChoiceDialog` und `wxMultiChoiceDialog` nützliche Klassen bereit.

wxColourDialog

Mit dem Dialog `wxColourDialog` kann der Anwender aus einer Standardpalette eine Farbe auswählen (siehe Abbildung 10.42).

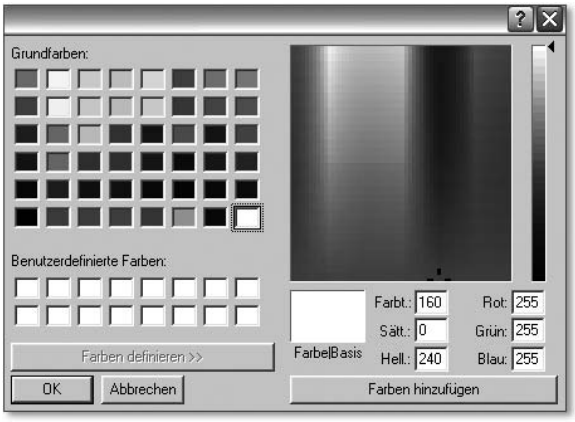

Abbildung 10.42 wxColourDialog

Die Syntax des Konstruktors zu `wxColourDialog` lautet:

```
wxColourDialog(wxWindow* parent, wxColourData* data = NULL);
```

Für `parent` geben Sie das Eltern-Fenster an. Mit `data` kann optional ein Zeiger auf die Farbpalette verwendet werden. Die Informationen von `wxColourData` werden benutzt, um die voreingestellten Farben festzulegen. Auch hier wird anschließend die Methode `ShowModal` auf `wxID_OK` überprüft. Wenn der Anwender die Farbauswahl bestätigt hat, können Sie die durch den Anwender modifizierten Daten mit der Methode `GetColourData` ermitteln:

```
wxColourData& GetColourData();
```

Hiermit werden die Farbdaten zurückgegeben, die mit dem Dialog assoziiert sind. In unserem Beispiel mit dem Text-Editor wurde dieser Dialog verwendet, um die Hintergrundfarbe des Textfelds zu verändern. Hierzu nochmals der Codeausschnitt:

```
// Option->Hintergrund
void TextFrame::OnMenuOptionBackgroundColor(
   wxCommandEvent &event)
{
   wxColourData colourData;
   wxColour colour = m_pTextCtrl->GetBackgroundColour();
   colourData.SetColour(colour);
   colourData.SetChooseFull(true);
   wxColourDialog *dlg = new wxColourDialog(
      this, &colourData);
   if ( dlg->ShowModal() == wxID_OK ) {
      colourData = dlg->GetColourData();
```

```
      m_pTextCtrl->SetBackgroundColour(
         colourData.GetColour());
      m_pTextCtrl->Refresh();
   }
   dlg->Destroy();
}
```

Auch `wxColourData` besitzt einige Methoden, die in diesem Dialog verwendet wurden. Mit `wxColourData::SetColour` legen Sie die vorgegebenen Farben fest, die anschließend bei der Farbauswahl angezeigt werden sollen (die Vorgabefarbe ist üblicherweise Schwarz oder Weiß). Mit `wxColourData::GetColour` können Sie diese Vorgabefarbe ermitteln.

Mit `wxColourData::SetChooseFull` geben Sie an, dass eine komplette Auswahl des Dialogs angezeigt werden soll. Diese Methode funktioniert allerdings nur unter MS Windows. Geben Sie diese Methode nicht mit an, sieht die Farbauswahl – anders als in Abbildung 10.42 – folgendermaßen aus:

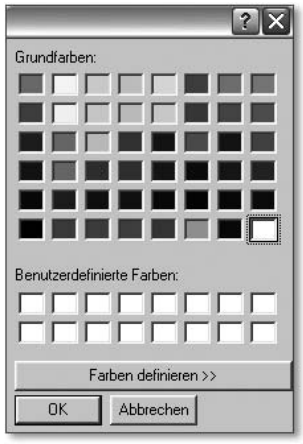

Abbildung 10.43 »wxColourDialog« ohne »wxColourData::SetChooseFull«

Im Text weiter oben war die Rede von vorgegebenen Farben. Mit der Methode `wxColourData::SetCustomColour` können Sie auch benutzerdefinierte Farben einstellen (siehe auch in Abbildung 10.43 das Text-Label »Benutzerdefinierte Farben:«). Wenn Sie dazu weitere Details benötigen, sollten Sie sich die Klassen `wxColourData`, `wxColour` und `wxColourDataBase` ansehen.

wxFontDialog

Mit der Klasse `wxFontDialog` kann der Anwender verschiedene Schriftarten auswählen. Auf einigen Plattformen kann hierbei auch gleich die Farbe der Schrift bestimmt werden (siehe Abbildung 10.44).

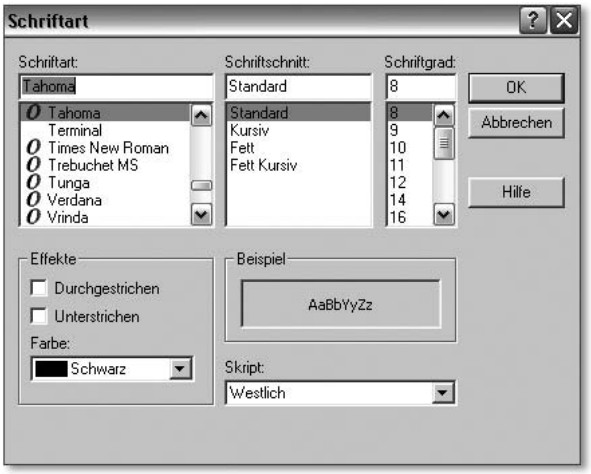

Abbildung 10.44 wxFontDialog

Der Dialog funktioniert ähnlich wie `wxColourDialog`. Zunächst der Konstruktor:

```
wxFontDialog(wxWindow* parent);
wxFontDialog(wxWindow* parent, const wxFontData& data);
```

Auch hierbei geben Sie zunächst mit `parent` das Eltern-Fenster an. Optional kann wieder mit `wxFontData data` ein Objekt verwendet werden, mit dem der Dialog initialisiert wird. Anschließend rufen Sie erneut die Methode `ShowModal` auf und überprüfen die Eingabe des Anwenders auf `wxID_OK`. Wenn der Anwender die Farbauswahl bestätigt hat, können Sie diese anwendermodifizierten Daten mit der Methode `GetFontData` ermitteln:

```
wxFontData& GetFontData();
```

Hiermit werden die Schriftdaten zurückgegeben, die mit dem Dialog assoziiert sind. Jetzt können Sie die Methode `wxFontData::GetChosenFont` (für die Schriftart) und gegebenenfalls `wxFontData::GetChosenColour` (für die Textfarbe) aufrufen. In unserem Beispiel mit dem Text-Editor wurde dieser Dialog verwendet, um die Schrift des Textfeldes zu verändern:

```
void TextFrame::OnMenuOptionFont(wxCommandEvent& event) {
   wxFontData fontData;
   wxFont font;
   wxColour colour;

   font = m_pTextCtrl->GetFont();
   fontData.SetInitialFont(font);
   colour = m_pTextCtrl->GetForegroundColour();
```

```
    fontData.SetColour(colour);
    fontData.SetShowHelp(true);

    wxFontDialog *dlg = new wxFontDialog(this, &fontData);
    if ( dlg->ShowModal() == wxID_OK ) {
       fontData = dlg->GetFontData();
       font = fontData.GetChosenFont();
       m_pTextCtrl->SetFont(font);
       m_pTextCtrl->SetForegroundColour(
          fontData.GetColour());
       m_pTextCtrl->Refresh();
    }
    dlg->Destroy();
}
```

wxSingleChoiceDialog

`wxSingleChoiceDialog` ist ein einfacher Dialog mit einer Liste von Strings, aus der der Anwender einen String auswählt.

Abbildung 10.45 wxSingleChoiceDialog

Die Syntax, mit der ein solcher Dialog konstruiert wird, lautet:

```
wxSingleChoiceDialog(
   wxWindow* parent,
   const wxString& message,
   const wxString& caption,
   int n,
   const wxString* choices,
   void** clientData = NULL,
   long style = wxCHOICEDLG_STYLE,
   const wxPoint& pos = wxDefaultPosition );
```

```
wxSingleChoiceDialog(
   wxWindow* parent,
   const wxString& message,
   const wxString& caption,
   const wxArrayString& choices,
   void** clientData = NULL,
   long style = wxCHOICEDLG_STYLE,
   const wxPoint& pos = wxDefaultPosition );
```

Hierbei gibt es wieder zwei Versionen – eine, die `n` Strings von `wxString` als Liste verwendet, und eine andere, die ein `wxArrayString` einsetzt. Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster
- `message` – die Nachricht, die im Dialog erscheint
- `caption` – die Nachricht in der Titelleiste
- `n` – die Anzahl der zur Auswahl stehenden Elemente
- `choices` – ein Array von Strings oder eine String-Liste mit der Auswahl
- `clientData` – ein Array mit Client-Daten, die mit dem Element verknüpft sind
- `style` – der Stil von `wxSingleChoiceDialog` (siehe Tabelle 10.73). Ohne Angabe gilt per Voreinstellung `wxDEFAULT_DIALOG_STYLE | wxRESIZE_BORDER | wxOK | wxCANCEL | wxCENTRE`.
- `pos` – die Position des Dialogs (nicht unter MS Windows gültig)

Style (wxSingleChoiceDialog)	Beschreibung
`wxOK`	Zeigt einen OK-Button an.
`wxCANCEL`	Zeigt einen Cancel-Button an.
`wxCENTRE`	Zentriert die Nachricht (2. Parameter).

Tabelle 10.73 Die Stile von »wxSingleChoiceDialog«

Zur Auswertung des Dialogs müssen Sie anschließend wieder die Eingabe des Anwenders mit `ShowModal` auf `wxID_OK` überprüfen. Die eigentliche Auswahl des Anwenders können Sie jetzt mit der Methode `GetSelection` (gibt den Index zurück) oder `GetStringSelection` (gibt den String zurück) auswerten. Mit welchem Wert der Dialog vorbelegt wird, können Sie wiederum mit den Gegenstücken `SetSelection` oder `SetStringSelection` angeben.

wxMultiChoiceDialog

`wxMultiChoiceDialog` funktioniert ähnlich wie `wxSingleChoiceDialog` und besitzt abgesehen vom Klassennamen auch dieselbe Syntax. Statt eines Strings können mit `wxMultiChoiceDialog` mehrere Strings auf einmal ausgewählt werden:

Abbildung 10.46 wxMultiChoiceDialog

Zur Auswertung des Dialogs müssen Sie wie gewohnt die Eingabe des Anwenders mit `ShowModal` auf `wxID_OK` überprüfen. Die eigentliche Auswahl des Anwenders können Sie jetzt mit der Methode `GetSelections` (gibt den Index zurück) auswerten. Mit welchem Wert der Dialog vorbelegt wird, können Sie wiederum mit `SetSelection` angeben.

Hierzu ein kurzes Listing, das die beiden Dialoge `wxSingleChoiceDialog` und `wxMultiChoiceDialog` in der Praxis demonstriert. Zunächst wieder die Header-Datei:

```
// base.h
#ifndef BASIC_H
#define BASIC_H

class wxFrameDemoApp : public wxApp {
   public: virtual bool OnInit();
};

class BasicFrame : public wxFrame {
private:
   wxMenuBar *MenuBar;
   wxMenu    *ExampleMenu;
   enum {
     MENU_DIALOG_SINGLE,
     MENU_DIALOG_MULTI
   };
   // Ereignis-Tabelle einrichten
   DECLARE_EVENT_TABLE()
public:
   BasicFrame( const wxChar *title,
               int xpos, int ypos,
               int width, int height);
   ~BasicFrame();
   // Methoden, die auf Ereignisse reagieren
```

```
   void OnMenuDialogSingleChoice (wxCommandEvent &event);
   void OnMenuDialogMultiChoice(wxCommandEvent &event);
};
#endif
```

Jetzt noch die Quelldatei:

```
// base.cpp
#include <wx/wx.h>
#include "base.h"
#include "wx/choicdlg.h"

IMPLEMENT_APP(wxFrameDemoApp)

bool wxFrameDemoApp::OnInit() {
   BasicFrame *frame = new BasicFrame(
      wxT("Demonstriert einige Dialoge"),
      50, 50, 300, 200);
   frame->Show(TRUE);
   SetTopWindow(frame);
   return TRUE;
}

BasicFrame::BasicFrame (
   const wxChar *title,
   int xpos, int ypos,
   int width, int height)
   : wxFrame ( (wxFrame *) NULL,
               -1, title,
               wxPoint(xpos, ypos),
               wxSize(width, height),wxDEFAULT_FRAME_STYLE)
{
   // Eine Menübar erzeugen
   MenuBar = new wxMenuBar();
   // Ein Menü erzeugen
   ExampleMenu = new wxMenu();
   ExampleMenu->Append(
      MENU_DIALOG_SINGLE, wxT("wxSingleChoiceDialog" ));
   ExampleMenu->Append(
      MENU_DIALOG_MULTI, wxT("wxMultiChoiceDialog") );
   MenuBar->Append( ExampleMenu, wxT("&Dialoge"));
   SetMenuBar(MenuBar);
   // Eine einfache Statusbar mit zwei Spalten erstellen
   CreateStatusBar(2);
}

BasicFrame::~BasicFrame() { }
```

```
// Ereignis-Tabelle
BEGIN_EVENT_TABLE(BasicFrame, wxFrame)
   EVT_MENU(MENU_DIALOG_SINGLE,
      BasicFrame::OnMenuDialogSingleChoice)
   EVT_MENU(MENU_DIALOG_MULTI,
      BasicFrame::OnMenuDialogMultiChoice)
END_EVENT_TABLE()

void BasicFrame::OnMenuDialogSingleChoice(
    wxCommandEvent &event) {
   wxArrayString choices;
   choices.Add(wxT("Auswahl 1"));
   choices.Add(wxT("Auswahl 2"));
   choices.Add(wxT("Auswahl 3"));
   choices.Add(wxT("Auswahl 4"));
   choices.Add(wxT("Auswahl 5"));

   wxSingleChoiceDialog dialog( this,
     wxT("Ein einfaches Beispiel von wxSingleChoiceDialog"),
     wxT("Bitte waehlen Sie aus:"), choices);

   dialog.SetSelection(1);
   if (dialog.ShowModal() == wxID_OK)
       wxMessageBox(
          dialog.GetStringSelection(),
          wxT("Sie haben gewählt"));
}

void BasicFrame::OnMenuDialogMultiChoice(
   wxCommandEvent &event) {
   wxArrayString choices;
   choices.Add(wxT("Auswahl 1"));
   choices.Add(wxT("Auswahl 2"));
   choices.Add(wxT("Auswahl 3"));
   choices.Add(wxT("Auswahl 4"));
   choices.Add(wxT("Auswahl 5"));
   wxMultiChoiceDialog dialog(this,
      wxT("Ein einfaches Beispiel von wxMultiChoiceDialog"),
      wxT("Bitte mehrere Auswahlen treffen"), choices);
   // Auswerten
   if (dialog.ShowModal() == wxID_OK) {
      wxArrayInt selections = dialog.GetSelections();
      wxString msg;
      msg.Printf(
         wxT("Sie haben %u Elemente ausgewaehlt:\n"),
         selections.GetCount());
```

```
    for( size_t n = 0; n < selections.GetCount(); n++ ) {
       msg += wxString::Format(
          wxT("%s\n"), choices[selections[n]].c_str());
    }
    wxMessageBox(msg, wxT("Sie haben gewählt"));
   }
}
```

Dialoge für die Anwender-Eingabe

Im Beispiel des Text-Editors wurden auch zwei Dialoge für die Eingabe von Zahlen und Strings verwendet. Hierbei handelte es sich um `wxNumberEntryDialog` (siehe Abbildung 10.47) und `wxTextEntryDialog` (siehe Abbildung 10.48).

Abbildung 10.47 wxNumberEntryDialog

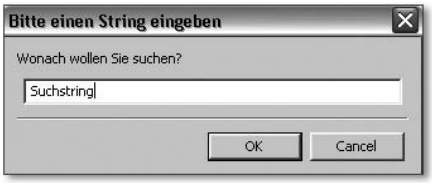

Abbildung 10.48 wxTextEntryDialog

wxNumberEntryDialog

Mit diesem Dialog können Sie den Anwender nach einer Ganzzahl in einem bestimmten Bereich fragen. Der Dialog verwendet zusätzlich einen Spin-Button, mit dem der Bereich auch mit der Maus bzw. den Pfeiltasten eingestellt werden kann.

Der Konstruktor hat folgende Syntax:

```
wxNumberEntryDialog(
   wxWindow* parent,
   const wxString& message,
   const wxString& prompt,
   const wxString& caption = "Please enter text",
   long defaultValue,
   long min,
```

```
   long max,
   long style = wxOK | wxCANCEL | wxCENTRE,
   const wxPoint& pos = wxDefaultPosition );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster
- `message` – die Nachricht, die im Dialog angezeigt wird
- `prompt` – der Text vor der Eingabebox
- `caption` – der Text in der Titelleiste
- `defaultValue` – der Vorgabewert
- `min` – der minimale Wert
- `max` – der maximale Wert
- `style` – der Stil von `wxNumberEntryDialog` (entspricht den `wxTextCtrl`-Stilen)
- `pos` – die Position des Dialogs

Nachdem Sie den Dialog erzeugt haben, müssen Sie wieder `ShowModal` aufrufen und die Eingabe auf `wxID_OK` überprüfen. Den numerischen Wert können Sie mit der Methode `GetValue` ermitteln. Hier nochmals der Codeausschnitt von `wxNumberEntryDialog`, der bereits beim Text-Editor verwendet wurde:

```
// Bearbeiten->Gehe zu Zeile
void TextFrame::OnMenuEditGoto(wxCommandEvent& event) {
   wxString s;
   s += wxString::Format(
       wxT("Eine Zeile zwischen 1 und %d auswaehlen"),
         m_pTextCtrl->GetNumberOfLines());

   wxNumberEntryDialog dialog(this,
      s,  wxT("Nummer eingeben: "),  wxT("Gehe zu Zeile"),
      1, 1, m_pTextCtrl->GetNumberOfLines());

   if (dialog.ShowModal() == wxID_OK) {
      long value = dialog.GetValue();
      m_pTextCtrl->SetInsertionPoint(
         m_pTextCtrl->XYToPosition(0, value-1));
   }
}
```

wxTextEntryDialog

Das Gegenstück zum Einlesen von Text ist `wxTextEntryDialog`. Die Syntax des Konstruktors lautet:

```
wxTextEntryDialog(
   wxWindow* parent,
   const wxString& message,
   const wxString& caption = "Please enter text",
   const wxString& defaultValue = "",
   long style = wxOK | wxCANCEL | wxCENTRE,
   const wxPoint& pos = wxDefaultPosition );
```

Die einzelnen Parameter haben folgende Bedeutung:

- `parent` – das Eltern-Fenster
- `message` – die Nachricht, die im Dialog angezeigt wird
- `caption` – der Text in der Titelleiste
- `defaultValue` – der Vorgabewert für das Textfeld. Hierbei kann ein numerischer Wert oder auch ein String verwendet werden.
- `style` – der Stil von `wxTextEntryDialog` (entspricht den `wxTextCtrl`-Stilen)
- `pos` – die Position des Dialogs

Nachdem Sie den Dialog erzeugt haben, wird wieder `ShowModal` aufgerufen und die Eingabe auf `wxID_OK` überprüft. Den eingegebenen String können Sie mit der Methode `GetValue` ermitteln – setzen können Sie einen String mit `SetValue`. Hier nochmals der Codeausschnitt von `wxTextEntryDialog`, der im Beispiel des Text-Editors verwendet wurde:

```
// Bearbeiten->Suchen
void TextFrame::OnMenuEditSearch(wxCommandEvent& event) {
   wxTextEntryDialog dialog(
      this,  wxT("Wonach wollen Sie suchen?"),
       wxT("Bitte einen String eingeben"),  wxT(""),
       wxOK | wxCANCEL );
   if (dialog.ShowModal() == wxID_OK) {
      // Such-String einlesen
      wxString s = dialog.GetValue();
      for( int line =1;
           line < m_pTextCtrl->GetNumberOfLines();
           ++line )  {
         bool search = true;
         // Komplette Zeile einlesen ...
         wxString tmp = m_pTextCtrl->GetLineText(line);
         // Anfangsposition des Suchstrings ermitteln
         int found = tmp.Find( s );
         if (found != -1) {
            int pos =
               m_pTextCtrl->XYToPosition(found,  line);
```

```
            m_pTextCtrl->SetSelection( pos, pos + s.Len());
            wxMessageDialog dialog(
               NULL,  wxT("Wollen Sie weitersuchen?"),
               wxT("Weitersuchen...?"),
               wxNO_DEFAULT|wxYES_NO|wxICON_QUESTION);

            switch ( dialog.ShowModal() ) {
               case wxID_YES: break;
               case wxID_NO:
                  search = false;
                  break;
               default: break;
            }
            if( !search )
               break;
         }
      }
   }
}
```

Weitere Dialoge

Natürlich gibt es noch zahlreiche weitere Dialoge in `wxWidgets`, die hier noch kurz erwähnt werden sollten.

- `wxPasswordEntryDialog` – diese Klasse entspricht in etwa `wxTextEntryDialog`, nur werden hier bei der Eingabe nicht die Zeichen, sondern Sternchen auf dem Bildschirm angezeigt. Dieser Dialog dient allerdings höchstens dazu, dass niemand das Passwort mitlesen kann.
- `wxFindReplaceDialog` – dieser Dialog kann eingesetzt werden, um nach einem Text zu suchen und diesen gegebenenfalls durch einen anderen Text zu ersetzen. Anstelle der Such-Funktion in unserem Text-Editor könnten Sie diese Klasse verwenden, die auch das Ersetzen mit anbietet.
- `wxPageSetupDialog` und `wxPrintDialog` – diese beiden Dialoge werden für das Drucken von Dokumenten aus einer Anwendung genutzt.

Natürlich sind nicht alle Dialoge für jede Anwendung geeignet. Wenn Sie spezielle Dialoge benötigen, können Sie auch selbst welche erstellen. Darauf wird hier allerdings nicht mehr näher eingegangen.

[»] **Hinweis**

Wo Sie noch mehr Informationen zur Erstellung von Dialogen und überhaupt zu `wxWidgets` finden, erfahren Sie am Ende des Kapitels.

10.3.15 Weitere Elemente und Techniken im Überblick

Zwar erscheint der Umfang dieses Kapitels recht groß, doch leider beschreibt es nur einen Bruchteil dessen, was `wxWidgets` wirklich kann. Was Sie bisher erfahren haben, sind lediglich die Grundlagen dieses wirklich umfangreichen Frameworks, mit dem Sie Programme aller Art und vor allem auf fast allen Plattformen erstellen können.

Was es noch gibt und wie Sie an weitere Informationen kommen, soll auf den nächsten Seiten noch erläutert werden.

Tastatur- und Maus-Eingabe

Bei den Anwendungen, die Sie bisher gesehen und eingesetzt haben, mussten Sie sich keine Gedanken um die Eingabe mit der Maus oder der Tastatur machen. Die Kommandos wie das Betätigen eines Menüelements oder Buttons haben das für uns übernommen und sind mit diesen Kontrollelementen eng verknüpft. Es ist allerdings auch möglich, selbständig auf die Ereignisse von Maus und Tastatur zu reagieren.

Wurde zum Beispiel bei einem Button ein Ereignis-Handle mit `EVT_BUTTON` eingerichtet und jemand betätigt diesen, so wird ein `wxCommandEvent`-Ereignis ausgelöst. Intern wurde dieses Ereignis von der Klasse `wxMouseEvent` ausgelöst, die mit `EVT_LEFT_DOWN` eingerichtet wird. Hierbei gibt es zahlreiche Ereignisse bei einer Maus, wofür wiederum viele Makros zum Einrichten eines Handles vorhanden sind. Mehr dazu finden Sie in der Dokumentation von `wxMouseEvent`.

Die Ereignisse der Tastatur werden von der Klasse `wxKeyEvent` repräsentiert. Hierbei gibt es im Grunde nur drei Arten von Ereignissen: Tastatur niederdrücken, Tastatur loslassen und einzelne Zeichen. Auch hierzu finden Sie weitere Informationen in der Dokumentation der Klasse `wxKeyEvent`.

Hinweis **[«]**

Es ist auch möglich, Joystick-Ereignisse zu behandeln. Für den Joystick selbst steht die Klasse `wxJoystick` und für die Ereignisse `wxJoystickEvent` zur Verfügung.

Device Context (Geräte-Kontext)

Alle Ausgaben von `wxWidgets` und auch von anderen GUI-Bibliotheken werden von einer grafischen Schnittstelle gesteuert (unter MS Windows ist das zum Beispiel GDI). Dies ist unabhängig vom gerade verwendeten Ausgabegerät, wie dem Bildschirm oder Drucker. Diese Schnittstelle arbeitet immer unabhängig vom Gerät, indem sie auf die Gerätetreiber der Ausgabegeräte zugreift. Sie merken schon, mit dem *Device Context* (kurz DC) greifen Sie auf eine sehr tiefe Ebene der

GUI-Programmierung zurück. Der Gerätetreiber des Ausgabegerätes selbst muss mindestens zwei Aktionen ausführen können, und zwar das Setzen eines Punkts und das Zeichnen einer Linie. Alle anderen Aktionen können dann softwaremäßig emuliert werden.

Die Schaltzentrale für alle Ausgaben, um auf die grafische Schnittstelle zuzugreifen, ist der Device Context. Dieser ist im Grunde nur eine Struktur, die Informationen dazu enthält, wie eine Text- und Grafikausgabe erfolgen muss. Bei jeder Ausgabe, die Sie auf dem Bildschirm machen, muss zunächst ein solcher DC geholt werden. Alle Ausgabefunktionen benötigen diesen DC (dies gilt übrigens nicht nur für `wxWidgets`). Wird der DC nicht mehr benötigt, muss er wieder freigegeben werden, da er eine Systemressource ist.

Theoretisch könnten Sie mit `wxDC` alles, vom Button bis zum Texteingabefeld, neu erfinden (bei `wxWidgets` wird hierbei das `wx` vorangestellt). Sie erhalten zum Beispiel einen sehr einfachen Button, indem Sie praktisch ein Rechteck zeichnen, eine Farbe setzen und einen Text hineinschreiben – hierbei findet allerdings noch keine Ereignis-Behandlung statt.

Zusammengefasst lässt sich sagen, dass alles, was mit der Ausgabe bzw. dem Zeichnen in `wxWidgets` zu tun hat (nicht nur bei `wxWidgets`), über einen Device Context geschieht. Sie verwenden immer eine Instanz einer Klasse, die von `wxDC` abgeleitet wurde. Beachten Sie, dass Sie hierbei nicht direkt auf dem Bildschirm oder Drucker zeichnen, sondern einen Device Context für den Bildschirm erzeugen und auf diesem zeichnen.

Weitere Informationen zum Device Context und dessen Methoden entnehmen Sie bitte der Dokumentation der Klasse `wxDC`.

Mit Bildern arbeiten

`wxWidgets` unterstützt auch einige Klassen für Bitmap-Bilder: `wxBitmap`, `wxIcon`, `wxCursor` und `wxImage`.

wxBitmap

Die Klasse `wxBitmap` kapselt das Konzept der plattformabhängigen Bitmap, mit dem auch Transparenz möglich ist. Unter MS Windows verwendet `wxBitmap` die geräteabhängige Bitmap (kurz DIB – *Device Independent Bitmap*). Bei GTK+ und X11 enthält jede Bitmap das entsprechende Pixmap-Objekt von GDK und X11. Unter Mac OS wird PICT verwendet.

`wxBitmap` wird hauptsächlich verwendet, um auf ein Fenster mit einem Device Context (`wxDC`) zu zeichnen oder um eine Bitmap als Label für Klassen wie zum

Beispiel wxBitmapButton, wxStaticBitmap oder wxToolBar zu verwenden (wie Sie es auf den vorangegangenen Seiten schon öfter gesehen haben).

wxIcon

Die Klasse wxIcon entspricht dem plattformtypischen Konzept eines Icons. Es handelt sich um ein kleines Bild mit Transparenz, das gewöhnlich verwendet wird, um einem Frame oder einem Dialog ein wiedererkennbares Aussehen zu geben. Unter GTK+, X11 und Mac OS ist ein Icon wieder nur eine Bitmap (wxBitmap), die immer mit einer Maske (wxMask) verknüpft ist, um das Bild mit Transparenz zu zeichnen. Unter MS Windows ist ein Icon ein HICON-Objekt.

Es ist natürlich auch möglich (und durchaus gängig), ein Icon direkt auf den Device Context mit der Methode wxDC::DrawIcon auszugeben. Auch weitere Widgets wie wxTreeCtrl, wxListCtrl oder wxNotebook können Sie in Verbindung mit der Klasse wxImageList mit Icons schmücken.

wxCursor

Mit wxCursor können Sie die Grafik des Mauszeigers ändern. Unter den verschiedenen Systemen ist allerdings sowohl das Aussehen als auch die Implementierung des Cursors unterschiedlich.

wxImage

wxImage ist die einzige plattformunabhängige Klasse, die mit Bildern arbeitet. Damit können Sie 24-Bit-Bilder (optional) mit Alpha-Kanal bearbeiten. Es lässt sich auch ein wxImage aus einem wxBitmap mit wxBitmap::ConvertToImage erzeugen. Ein wxImage kann aber auch von einer Datei geladen werden. Dabei stehen Ihnen viele Grafikformate zur Verfügung (zum Beispiel BMP, JPG, GIF, TIFF, PNG usw.). Auch das Konvertieren in andere Formate ist möglich. wxImage dient dazu, Bilder zu bearbeiten, zu verändern und wieder abzuspeichern. Allerdings können Sie ein wxImage nicht mehr direkt auf ein Device Context zeichnen. Hierzu können Sie allerdings wieder aus einem wxImage ein wxBitmap machen.

Zwischenablage und Drag & Drop

Das Herzstück der Zwischenablage und des Drag & Drop ist die Klasse wxDataObject. Alle Objekte dieser Klasse bzw. diejenigen, die von dieser Klasse abgeleitet wurden, repräsentieren die Daten, die mit der Maus gezogen und abgelegt werden oder in die Zwischenablage kopiert und dort wieder herausgeholt werden können.

Logischerweise finden Sie für die Zwischenablage und das Drag & Drop Operationen in zweifacher Ausführung: eine für die Quelle und eine weitere für das Ziel oder genauer, eine für den Datenprovider und eine für den Datenempfänger.

Für die Zwischenablage gibt es die Klasse `wxClipboard`. Wenn Sie diese Zwischenablage verwenden wollen, müssen Sie die darin definierten Methoden mit dem globalen Objekt `wxTheClipboard` verwenden.

Beim Drag & Drop können Sie entweder das Drag (Quelle), das Drop (Ziel) oder beides implementieren. Hierfür sind die Klassen `wxDropSource` und `wxDropTarget` und deren darin implementierte Methoden verantwortlich. Wenn Sie beispielsweise wollen, dass aus einer anderen Anwendung Text in Ihre Anwendung gezogen werden kann, müssen Sie das Drop implementieren. Wenn Ihnen außerdem beim Drag & Drop der Mauscursor nicht genug anzeigt, können Sie mit `wxDragImage` eine andere Grafik bestimmen, die beim Ziehen von Objekten mit der Maus angezeigt werden soll.

Fortgeschrittene Fensterklassen

`wxWidgets` bietet natürlich auch noch viel komplexere Fensterklassen als die bisher beschriebenen.

- `wxTreeCtrl` – diese Baumkontrolle zeigt Informationen als eine Hierarchie mit Elementen an, die expandiert und wieder zusammengefahren werden kann (siehe Abbildung 10.49).
- `wxListCtrl` – diese Listenkontrolle zeigt Elemente auf vier verschiedene Arten an: als mehrzeilige Listenansicht, als mehrspaltige Reportansicht mit optionalen Icons, als Ansicht mit großen Icons und als Ansicht mit kleinen Icons (siehe Abbildung 10.50).
- `wxWizard` – diese Klasse wird für typische Installations-Dialoge (auch als Installer bekannt) verwendet, wie sie MS Windows-Anwender von der Installation einer Software her kennen dürften. Hierbei handelt es sich um eine Sequenz von mehreren Dialog-Fenstern, wo der Anwender über die Buttons Next zum nächsten Dialog und Previous zum vorherigen Dialog schalten kann. Es gibt auch einen Button Cancel, um den Vorgang abzubrechen (siehe Abbildung 10.51). Am Ende wird diese Sequenz mit dem Button Finish beendet.
- `wxHtmlWindow` – diese Klasse wird gewöhnlich vom eingebauten Hilfssystem von `wxWidgets` verwendet. Aber sie lässt sich auch gut einsetzen, um in Ihrer Anwendung einen formatierten Text und/oder Grafiken anzuzeigen. Dabei sind HTML-Elemente wie Tabellen (keine Frames), animierte GIFs, Links (unterstrichen), Schriftarten, Hintergrundfarbe, Ausrichtung von Text und Grafiken usw. möglich. Leider ist hierbei kein CSS möglich. Mit angepassten Tags kann aber ein ähnlicher Effekt erzielt werden. Sie können hiermit praktische einfache HTML-Seiten in einem Fenster anzeigen lassen, die auch in die Zwischenablage kopiert werden können (siehe Abbildung 10.52).

- `wxGrid` – diese Klasse ist eine sehr umfangreiche Kontrolle, die sich zur Anzeige von Daten in Tabellenform, wie beispielsweise Excel, anbietet (siehe Abbildung 10.53).
- `wxTaskBarIcon` – wollen Sie dem Anwender die Möglichkeit einräumen, die Anwendung auch im System-Tray oder ähnlich wiederzugeben, dann kann diese Klasse dazu verwendet werden.

Hinweis [«]

Die folgenden Beispiele zu den Abbildungen finden Sie mit Ihrer wxWidgets-Bibliothek im Verzeichnis *Examples* – und natürlich auch auf der Buch-CD.

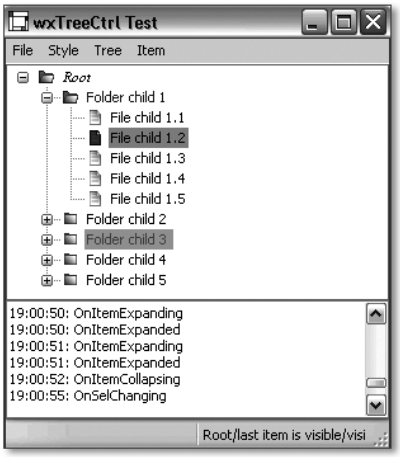

Abbildung 10.49 wxTreeCtrl

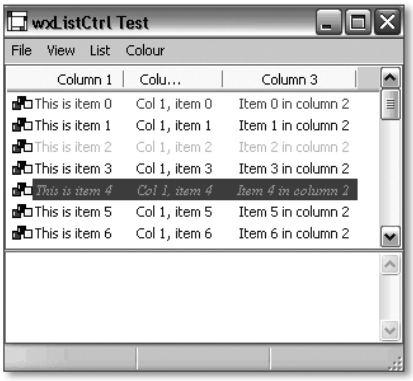

Abbildung 10.50 wxListCtrl

Abbildung 10.51 wxWizard

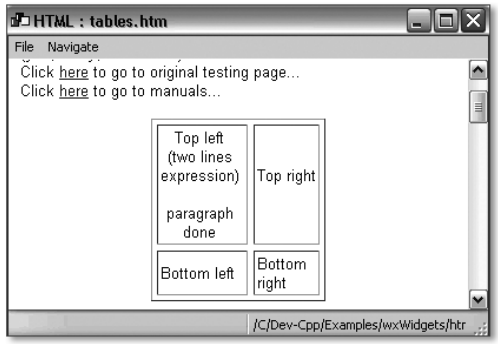

Abbildung 10.52 wxHtmlWindow

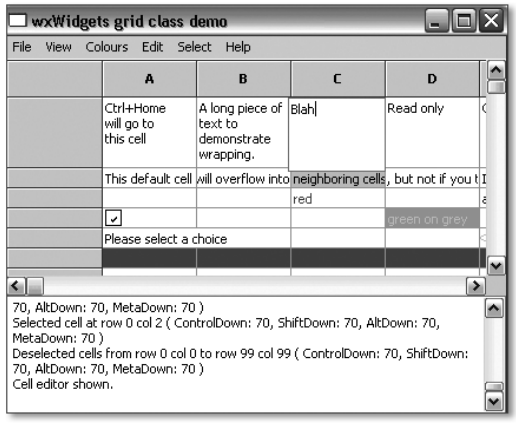

Abbildung 10.53 wxGrid

Datenstrukturen

Das Speichern und Bearbeiten von Daten ist immer der wichtigste Teil einer Anwendung. `wxWidgets` hat auch hierbei eigene Klassen wie zum Beispiel Strings, Arrays, verkettete Listen, Hashmaps, Datum und Uhrzeit in der Bibliothek implementiert.

Hier stellt sich die Frage, warum in `wxWidgets` viele Datenstrukturen implementiert wurden, die man auch von STL hätte verwenden können. Der Grund ist folgender: `wxWidgets` gibt es schon seit 1992 – STL zwar auch, aber damals war STL noch nicht für alle Plattformen und Compiler erhältlich. Aber es ist trotzdem möglich, STL in den `wxWidgets`-Anwendungen einzusetzen. Hierzu müssen Sie in der Anwendung `wxUSE_STL` der Header-Datei *<setup.h>* auf `1` setzen.

Strings (»wxString«)

Die Klasse `wxString` entspricht zu 90 % den Strings der Standardbibliothek (`std::string`). Somit werden Sie fast alle Methoden unter denselben oder ähnlichen Namen, wie Sie sie bereits in Abschnitt 7.1, »Die String-Bibliothek (string-Klasse)«, kennengelernt haben, auch in `wxString` wiederfinden. `wxString` bietet allerdings noch einiges mehr als die `std::string`. Beispielsweise unterstützt `wxString` vollständig Unicode und enthält Methoden, um von ANSI in Unicode und umgekehrt zu konvertieren. Weitere Klassen im Zusammenhang mit den Strings, die hier auch erwähnt werden sollen, sind:

- `wxStringTokenizer` – mit dieser Klasse können Sie einen String mit einem Token aufteilen. Standardmäßig wird ein String durch ein Leerzeichen aufgetrennt, was aber auch verändert werden kann.
- `wxRegEx` – natürlich bietet auch `wxWidgets` mit der Klasse `wxRegEx` die regulären Ausdrücke zum Suchen und Ersetzen an.

Arrays (»wxArray«)

Das dynamische Array von `wxWidgets` ist mit der Klasse `wxArray` implementiert. Wenn neue Elemente hinzugefügt werden sollen und nicht mehr genug Speicher vorhanden ist, wird automatisch neuer Speicher reserviert. Auch eine Überprüfung des Bereichs (range checking) ist hierbei implementiert. Natürlich ist auch hier – wie bei einem C-Array – die Zugriffszeit konstant.

Dabei sind insgesamt drei verschiedene Arten von Arrays in `wxWidgets` implementiert. Alle diese Arrays sind von der Basisklasse `wxBaseArray` abgeleitet, die an keinen Typ gebunden ist und nicht direkt verwendet werden kann. Die drei Klassen sind:

- `wxArray` – diese Klasse wird verwendet, um Integer-Typen und Zeiger zu speichern. Allerdings sollte man beachten, dass mit dieser Klasse nur integrale Typen (`bool`, `char`, `short`, `int`, `long` und ihre `unsigned`-Varianten) oder Zeiger verwendet werden können. Datentypen wie `float` oder `double` sollten nicht in `wxArray` gespeichert werden.
- `wxSortedArray` – diese Klasse ist im Grunde eine Erweiterung von `wxArray` mit all ihren Eigenschaften. Dieses Array sollten Sie im Gegensatz zu `wxArray` verwenden, wenn Sie regelmäßig in einem Array nach Elementen suchen. Diese Klasse benötigt allerdings eine Funktion, die zwei Array-Elemente vom selben Typ miteinander vergleicht und die Elemente immer in einer sortierten Reihenfolge speichert. Der Vorteil von `wxSortedArray` gegenüber `wxArray` liegt in der Suche nach Elementen im Array, da in einem sortierten Array erheblich schneller ein Element gefunden werden kann. Allerdings gilt auch hier dieselbe Einschränkung wie bei `wxArray`, dass nur integrale Typen und Zeiger gespeichert werden können.
- `wxObjArray` – mit dieser Klasse werden die einzelnen Elemente wie Objekte behandelt.

Eine weitere Form von Array ist die Klasse `wxArrayString`, die `wxString`-Objekte mit denselben Fähigkeiten wie die anderen `wxArray`-Klassen speichert. Alle Methoden aus den anderen `wxArray`-Klassen stehen auch in der Klasse `wxArrayString` zur Verfügung.

Verkettete Listen (»wxList«)

Die Klasse `wxList` verkörpert eine doppelt verkettete Liste, die beliebige Typen speichern kann. Abhängig davon, welcher Konstruktor eingesetzt wird, kann auch ein Schlüssel in Form eines Integers oder eines Strings verwendet werden, um eine einfache Loopup-Fähigkeit zu nutzen. Allerdings sollte man für solche Fälle eher auf `wxHashMap` zurückgreifen. Die Klasse `wxList` benötigt außerdem die abstrakte Klasse `wxNode`. Wenn Sie zum Beispiel eine neue Liste definieren, wird auch ein neuer Knotentyp, abgeleitet von `wxNodeBase`, erzeugt.

wxHashMap

`wxHashMap` ist eine einfache, typsichere und auch effektive Hashmap-Klasse, die eine fast identische Schnittstelle wie der entsprechende STL-Container (siehe Abschnitt 5.3.5, »Container«) besitzt (`std::map` und `std::hash_map`).

Datum und Uhrzeit

`wxWidgets` besitzt mit der Klasse `wxDateTime` eine reichhaltige Klasse für Informationen zu Datum und Uhrzeit mit sehr vielen Methoden wie das Formatieren der Angaben, Zeitzonen, Arithmetik mit Zeit und Datum und noch einiges mehr.

Weitere Hilfsklassen hierzu sind `wxTimeSpan` und `wxDateSpan`, die einen sehr bequemen Weg anbieten, um ein existierendes `wxDateTime`-Objekt zu verändern.

Hilfsklassen für Datenstrukturen

Einige dieser Hilfsklassen wurden schon oft in den Beispielen verwendet, ohne dass näher darauf eingegangen wurde. Daher sollen im Folgenden ein paar dieser Hilfsklassen beschrieben werden.

- `wxObject` – diese Klasse ist die Basisklasse für alle `wxWidgets`-Klassen und stellt Folgendes bereit: Typinformationen, Referenz-Zähler, virtuelle Destruktor-Deklaration und (optional) Debugging-Versionen von `new` und `delete`. Die Klasse `wxClassInfo` wird genutzt, um Metadaten über die Klassen zu speichern; sie wird auch von einigen `wxObject`-Methoden verwendet.
- `wxLongLong` – diese Klasse repräsentiert eine 64 Bit lange Nummer.
- `wxPoint` – diese Klasse wird eingesetzt, um eine bestimmte Position in einem Fenster oder auf dem Bildschirm anzugeben. Sie verwendet die Koordinaten als x/y-Paar. Da diese beiden Daten als `public` deklariert sind, hat man jederzeit direkten Zugriff auf x und y.
- `wxRect` – diese Klasse wird für Veränderungen von rechteckigen Informationen von `wxWidgets` verwendet. `wxRect` enthält zwei x/y-Paare, eines für die Höhe und eines für die Breite. Auch hier sind die Eigenschaften als `public` deklariert, so dass man jederzeit Zugriff darauf hat.
- `wxSize` – diese Klasse wird in `wxWidgets` verwendet, wenn die Größe für Fenster, Kontrollelemente usw. angegeben wird.

Datei-, Verzeichnisklassen

`wxWidgets` unterstützt auch eine ganze Auswahl an Methoden für ein plattformunabhängiges File-Handling, die in den Klassen `wxFile` und `wxFFile` definiert sind. `wxFile` wird für die Ein-/Ausgabe auf der niedrigeren (Filedeskriptor-) Ebene und `wxFFile` für die höhere (`FILE`-Zeiger-)Ebene verwendet.

Wenn Sie kleinere Dateien Zeile für Zeile lesen oder schreiben wollen, können Sie die Klasse `wxTextFile` und deren Methoden anwenden.

Die von `wxFile` abgeleitete Klasse `wxTempFile` wird genutzt, um Daten in eine temporäre Datei zu schreiben, die erst in die aktuelle Datei geschrieben wird, wenn der Anwender ein `Commit` aufruft.

Mit der Klasse `wxDir` haben Sie ein portables Gegenstück zu den UNIX-Funktionen `open`, `read` und `closedir`, mit denen Sie den Inhalt eines Verzeichnisses auf-

listen können. Natürlich wurde auch eine Methode implementiert (`Traverse`), mit der Sie einen ganzen Verzeichnisbaum rekursiv durchlaufen können.

Die Klasse `wxFileName` stellt den Namen einer Datei dar. Mit dieser Methode können Sie einen Dateinamen parsen, neue Dateien oder Verzeichnisse anlegen, überprüfen, ob eine Datei oder ein Verzeichnis existiert, Arbeitsverzeichnisse ermitteln und vieles mehr.

Stream-Klassen

`wxStreamBase` ist die Basisklasse für alle anderen Stream-Klassen. Die beiden Klassen `wxOutputStream` und `wxInputStream` wiederum sind der Grundstein für die weiteren Klassen zum Schreiben oder Lesen. Hierzu ein kurzer Überblick über die vielen Stream-Klassen von `wxWidgets`:

- `wxFileInputStream` und `wxFileOutputStream` – beide Klassen basieren auf der `wxFile`-Klasse und können mit einem Dateinamen, einem `wxFile`-Objekt oder einem Filedeskriptor initialisiert werden.
- `wxFFileInputStream` und `wxFFileOutputStream` – wie `wxFileInputStream` und `wxFileOutputStream`, nur basieren diese Klassen auf der `wxFFile`-Klasse, also der höheren Ebene.
- `wxMemoryInputStream` und `wxMemoryOutputStream` – diese Klassen nutzen einen internen Puffer des Speichers für den Datenfluss. Der Konstruktor beider Klassen verwendet einen `char*`-Puffer und die Größe des Puffers, die dynamisch vergrößert werden kann.
- `wxStringInputStream` und `wxStringOutputStream` – `wxStringInputStream` verwendet eine `wxString`-Referenz, von der die Daten gelesen werden sollen, und `wxStringOutputStream` nutzt hierzu einen `wxString`-Zeiger, der angibt, wohin der Stream die Daten ausgeben soll.
- `wxTextInputStream` und `wxTextOutputStream` – beide Klassen arbeiten auf einer höheren Ebene und lesen bzw. schreiben Daten von einer bzw. in eine Datei in einer lesbaren Form.
- `wxDataInputStream` und `wxDataOutputStream` – ähnlich wie `wxTextInputStream` und `wxTextOutputStream`, nur wird hier mit binären Daten gearbeitet. Der Vorteil dieser Streams ist, dass die Daten in einer portablen Form von einer Plattform zur anderen fließen.
- `wxSocketOutputStream` und `wxSocketInputStream` – beide Klassen werden in Verbindung mit einem `wxSocket`-Objekt initialisiert.
- `wxFilterInputStream` und `wxFilterOutputStream` – diese Klassen sind Basisklassen für Streams, die auf anderen Streams basieren, beispielsweise die

Klasse `wxZlibInputStream`. Wenn Sie hier einen Eingabe-Stream mit einer *zlib*-Datei verwenden, können Sie die Daten lesen, ohne diese Datei zu dekomprimieren.

Multithreading

Gerade in einer Zeit, in der Prozessoren mit einem Dual-Core für den Massenmarkt angeboten werden, erscheint die Möglichkeit der Verwendung von Multithreads auch für Privatanwender sehr interessant. Häufig hat die zweite Einheit des Prozessors auf einem Rechner nichts zu tun. Die Verwendung des zweiten Prozessor-Kerns hängt nicht vom Rechner ab, sondern von der Software. Hierfür würde es sich anbieten, dass der eine Teil des Prozessors die Ereignisse des Anwenders abfängt und der andere Teil währenddessen ein Fenster neu zeichnet oder Berechnungen ausführt. `wxWidgets` bietet Ihnen mit der Klasse `wxThread` die Möglichkeit, mehrere Aktionen in einer Anwendung gleichzeitig auszuführen und (vor allem auch) zu synchronisieren.

Sockets (»wxSocket«)

Natürlich gibt es auch Socket-Operationen in `wxWidgets`. All diese Operationen bauen auf der Basisklasse `wxSocketBase` auf, die alle grundlegenden Funktionen zum Senden und Empfangen von Daten, das Schließen von Sockets, den Fehlerbericht und noch vieles mehr enthält. Abhängig davon, ob Sie an einem Socket lauschen oder eine Verbindung mit einem Server eingehen, sind hierzu die Klassen `wxSocketServer` und `wxSocketClient` definiert. Ereignisse auf ein Socket werden mit der Klasse `wxSocketEvent` ausgewertet und bearbeitet. Mit den bereits erwähnten Streams `wxSocketInputStream` und `wxSocketOutputStream` können Sie eine Verbindung mit anderen Streams eingehen, um die Daten vom bzw. zum Socket zu transformieren.

Die Klasse `wxSocket` ist aber noch viel flexibler und nicht nur auf die Kommunikation zwischen zwei Prozessen über Sockets beschränkt. Wenn Sie beispielsweise FTP- oder HTTP-Operationen durchführen wollen, können Sie die Klassen `wxFTP` oder `wxHTTP` nutzen, die ebenfalls `wxSocket` verwenden.

Hinweis **[«]**

`wxWidgets` unterstützt auch eine höhere Ebene der Interprozess-Kommunikation mit den Klassen `wxServer`, `wxClient` und `wxConnection`, die auf Microsofts DDE-(*Dynamic Data Exchange*-)Protokoll aufbauen. Allerdings ist diese Möglichkeit auf MS Windows beschränkt. Sie können diese Klassen zwar auch auf anderen Plattformen verwenden, aber hierbei wird (intern) auf die Sockets und nicht auf DDE zurückgegriffen.

MFC versus »wxWidgets«

Der Platzhirsch beim Schreiben von GUI-Anwendungen unter MS Windows ist nach wie vor MFC, aber wenn es darum geht, Anwendungen für mehrere Plattformen zu erstellen, ist `wxWidgets` eine sehr interessante (und die beste) Alternative. Soll eine Anwendung, von der Sie den MFC-Code besitzen, in eine `wxWidgets`-Anwendung umgeschrieben werden, damit diese auch auf anderen Plattformen zur Verfügung steht, dann ist dies einfacher, als Sie vielleicht denken, weil `wxWidgets` viele Konzepte und Konstruktionen besitzt, die denen von MFC ähneln.

Unicode

Sie sollten hierzu auf jeden Fall in der Dokumentation den Punkt »Unicode support in wxWidgets« lesen. Als schnelle Referenz sollten Sie hierbei folgende Aspekte beachten (wenn möglich):

- Verwenden Sie immer `wxChar` statt `char`.
- Schließen Sie ein konstantes String-Literal immer zwischen dem Makro `wxT()` oder `_T()` (beides ist dasselbe) ein. Wollen Sie dieses String-Literal übersetzen (zum Beispiel mit gettext), dann sollten Sie es zwischen `_()` einschließen.
- Verwenden Sie immer `wxString` statt Strings im C-Stil.

Ausblick

Natürlich bietet diese Einführung nur einen kleinen Ausschnitt aus dem wirklich mächtigen Framework. Wenn Sie jetzt noch intensiver in die GUI-Programmierung mit `wxWidgets` einsteigen wollen, sollten Sie sich die neueste Dokumentation dazu besorgen und sich dann mit den vielen Codebeispielen auseinandersetzen, die `wxWidgets` liefert.

Anschließend sollten Sie sich ein RAD-Tool aussuchen, mit dem Sie sich die ganzen GUI-Elemente zusammenklicken können. Für `wxWidgets` stehen mittlerweile viele solcher Tools zur Verfügung. Beispielsweise gibt es eine Version, die im Bloodshed-Dev-C++-Compiler (`wxDev-C++`) integriert ist. Sie finden diese Version auch auf der Buch-CD. Ein weiteres, sehr vielversprechendes (und kostenloses) RAD-Tool für `wxWidgets` finden Sie bei der Entwicklungsumgebung `Code::Blocks` mit `wxSmith`. Zu erwähnen wären hier auch noch `wxGlade` und `wxDesigner`, wobei Letzteres ein kommerzielles RAD ist, meines Erachtens aber den besten Eindruck macht.

Natürlich bedeuten diese RAD-Anwendungen nicht, dass sich jeder unerfahrene User seine Anwendungen zusammenklicken kann. Wenn es um die Details geht, kommt man um das Grundwissen der `wxWidgets`-Programmierung nicht herum. Aber die RAD-Tools beschleunigen den Entwicklungsprozess der Software erheblich.

Kein Buch ohne Anhang. Neben den üblichen Tabellen zu den Operatoren, der Vorrangtabelle und den Schlüsselwörtern in C++ finden Sie hier zusätzlich Wissenswertes zur Informationsspeicherung (Zahlensysteme, Bits und Bytes) und zu Zeichensätzen.

A Anhang

A.1 Operatoren in C++ und deren Bedeutung (Übersicht)

Hier finden Sie eine Zusammenfassung der Operatoren von C++ und deren Bezeichnung bzw. Verwendung.

Operator	Bezeichnung
Arithmetische Operatoren	
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo-Division
+	Vorzeichenoperator (unär)
-	Vorzeichenoperator (unär)
++	Inkrementoperator
--	Dekrementoperator
Vergleichsoperatoren	
==	gleich
!=	ungleich
<	kleiner
<=	kleiner gleich
>	größer
>=	größer gleich
Logische Operatoren	
&&	UND
||	ODER
!	NICHT

Operator	Bezeichnung
Zuweisungsoperatoren	
`=`	eine einfache Zuweisung
`op=`	zusammengesetzte Zuweisung
Bit-Operatoren	
`&`	UND
`~`	NICHT
`|`	ODER
`^`	Exklusiv-ODER
`>>`	Shift-Operator
`<<`	Shift-Operator
Cast-Operatoren	
`(typ)`	C-Cast
`dynamic_cast<>`	dynamischer Cast
`static_cast<>`	statischer Cast
`const_cast<>`	Const-Cast
`reinterpret_cast<>`	Reinterpret-Cast
Zugriffsoperatoren	
`::`	Bereichsoperator
`[ ]`	Indexoperator
`*`	Verweisoperator
`.`	Punktoperator
`->`	Pfeiloperator
`.*`	Dereferenzierung von Elementzeigern
`->*`	Dereferenzierung von Elementzeigern
Operatoren für die Speicherverwaltung	
`new`	ein Objekt dynamisch anlegen
`delete`	ein dynamisch angelegtes Objekt zerstören
`new []`	Klassen-Array dynamisch anlegen
`delete []`	ein dynamisch angelegtes Klassen-Array zerstören
Restliche Operatoren	
`?:`	Auswahloperator (ternär)
`,`	Kommaoperator
`name()`	Aufruf der Funktion `name`
`typ()`	ein temporäres Objekt vom Typ `typ` erzeugen
`sizeof()`	Größe eines Datentyps ermitteln
`typeid()`	Typinformationen

A.2 Vorrangtabelle der Operatoren

In der folgenden Tabelle werden die Operatoren von C++ und ihre Assoziativität (die Bindung der Operanden) in absteigender Reihenfolge aufgelistet. Operatoren derselben Prioritätsklasse haben dieselbe Rangstufe.

Priorität	Operator	Assoziativität
1.	`::`	von links nach rechts

2.	`. -> []`
	`++` (Postfix) `--` (Postfix)
	`name()`, `typ()`, `typeid()`
	`dynamic_cast<>`, `static_cast<>`
	`const_cast<>`, `reinterpret_cast<>`
		von links nach rechts

3.	`! ~`
	`+` (unär) – (unär)
	`++` (Präfix) `--` (Präfix)
	`& *` (Verweis)
	`new`, `new[]`
	`delete`, `delete[]`
	`(typ)`, `sizeof()`
		von rechts nach links

4.	`.* ->*`	von links nach rechts
5.	`* / %`	von links nach rechts
6.	+ (binär) – (binär)	von links nach rechts
7.	`<< >>`	von links nach rechts
8.	`< <= => >`	von links nach rechts
9.	`== !=`	von links nach rechts
10.	`&`	von links nach rechts
11.	`^`	von links nach rechts
12.	`|`	von links nach rechts
13.	`&&`	von links nach rechts
14.	`||`	von links nach rechts
15.	`?:`	von rechts nach links

16.	`= += -=`
	`*= /= %=`
	`&= ^= |=`
	`<<= >>=`
		von rechts nach links

17.	`,`	von links nach rechts

A.3 Schlüsselwörter von C++

Die Schlüsselwörter von C++ werden in zwei Tabellen dargestellt. In einer Tabelle finden Sie die Schlüsselwörter von C und C++ und in einer weiteren Tabelle die Schlüsselwörter von C++ allein, ohne C (fett gesetzte Schlüsselwörter wurden erst mit dem ANSI-C99-Standard eingeführt).

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	**inline**	int	long	register
restrict	return	short	signed	sizeof
static	struct	switch	typedef	union
unsigned	void	volatile	while	**_Bool**
_Complex	**_Imaginary**			

Tabelle A.1 Schlüsselwörter von C und C++

and	dynamic_cast	operator	true
and_eq	explicit	or	try
Asm	export	or_eq	typeid
bitand	false	private	typename
bitor	friend	protected	using
Bool	inline	public	virtual
catch	mutable	reinterpret_cast	wchar_t
class	namespace	static_cast	xor
compl	new	template	xor_eq
const_cast	not	this	
delete	not_eq	throw	

Tabelle A.2 Reine C++-Schlüsselwörter

A.4 Informationsspeicherung

Im Allgemeinen macht man sich keine Gedanken über den Begriff *Information* – aber was versteht man eigentlich darunter? Eine Information wird mit Hilfe von Nachrichten übertragen. Eine solche Nachricht besteht gewöhnlich aus einer Folge von Zeichen, die durch Signale dargestellt werden. Bestimmt wird eine Information durch die Größe, durch die Neuigkeit und somit auch durch die

Unwahrscheinlichkeit der übertragenen Zeichen. Hierzu wurde von Shannon folgende formale Definition aufgestellt:

I(X) = -log2 p(X)

Hierbei ist X eine Nachricht und p(X) die Wahrscheinlichkeit ihres Auftretens.

Ich muss niemandem mehr erzählen, dass Computer ihre Informationen als Folge von Einsen (1) und Nullen (0) abspeichern. Man bezeichnet diese Form der Darstellung als *binäre Codierung*. Ein Beispiel dieser binären Darstellung ist das *Dualsystem* (auch *Zweiersystem* genannt).

Bezogen auf die Definition von Shannon und die binäre Darstellung, bedeutet dies, dass die Wahrscheinlichkeit der 0 und der 1 gleich 0,5 ist. Somit definiert man hierbei:

I(0) = I(1) = 1 Bit

Die einzelnen Binärstellen, die der Computer speichert, werden als Bit (*BInary digiT*, dt.: Binärziffer) bezeichnet. Ein Bit ist somit die kleinste Informationseinheit, mit der der Computer arbeiten kann. Im Grunde ist ein Bit die einzige vorstellbare Möglichkeit, um überhaupt in einem Computer Zeichen darzustellen.

Hinweis **[«]**

In der Tat gibt es auch noch ein System, das lediglich mit einem Symbol auskommt – man spricht hierbei vom *Unärsystem*. Das Unärsystem ist ein Additionssystem, das für einfache Zählaufgaben verwendet werden kann. Das Erhöhen einer Zahl um den Wert 1 geschieht hierbei durch das Anhängen eines weiteren Symbols. Dieses Unärsystem findet man z. B. als Strichliste, wo häufig aus Gründen der einfacheren Darstellung jeder fünfte Strich quer durch die vier vorherigen gezogen wird. Die entstehende Zahl ist so in Fünferblöcke gruppiert und damit leichter lesbar. Aber spätestens zur Darstellung der Lücke zwischen zwei auf diese Weise codierten Symbolen wäre dann doch wieder eine andere Informationsart erforderlich.

A.4.1 Zahlensysteme

Um zu verstehen, wie die Speicherung von Werten im Computer vor sich geht, benötigen Sie die Kenntnisse verschiedener Zahlensysteme.

Bevor man sich den einzelnen Zahlensystemen zuwenden kann, muss man auch wissen, was ein *Stellenwertsystem* ist. Bei einem Stellenwertsystem hängt der Wert einer einzelnen Ziffer von dem Eigenwert und ihrer Position in der Zahl ab. Jede Stelle hat dabei einen festen Grundwert, mit dem der Wert der einzelnen Ziffer multipliziert wird. Eine Basis B gibt an, wie viele verschiedene Ziffern nötig sind, denn der Grundwert jeder Stelle ist das B-fache der rechts von ihr befindli-

chen Stelle. Dadurch werden die Ziffern von 0 bis B-1 benötigt, um alle möglichen ganzen Zahlen darzustellen. Der Wert B selbst wird durch eine 1 auf der nächsthöheren Stelle ausgedrückt.

Dezimalsystem

Das *Dezimalsystem* oder *Zehnersystem* (lat.: decimus = der Zehnte) ist ein Stellenwertsystem zur Darstellung von Zahlen. Es ist heute das weltweit am stärksten verbreitete Zahlensystem und stammt ursprünglich aus Indien. Im Grunde hat das Dezimalsystem mit Computern am wenigsten zu tun. Allerdings wird dieses Zahlensystem gewöhnlich im Alltag verwendet. Zur Ein- bzw. Ausgabe von Zahlen am Computer wird diese Form vorwiegend verwendet.

Das Dezimalsystem verwendet als Basis B=10, so dass der Wert der ersten Stelle (rechts) 1 ist und sich mit jeder weiteren Stelle verzehnfacht. Die Dezimalzahl 2345 wird beispielsweise formal folgendermaßen analysiert:

Ziffern	2	3	4	5
Stellenwert	1000	100	10	1
Schema	10^3	10^2	10^1	10^0
Gesamtwert	2000	300	40	5

Tabelle A.3 Darstellung einer Dezimalzahl

Dualsystem (Binärsystem)

Das *Dualsystem* (auch *Binärsystem* oder *Zweiersystem*) ist für den Computer selbst das wichtigste, weil dieser intern damit arbeitet. Das Dualsystem (lat.: dualis = zwei enthaltend) ist ein Zahlensystem aus nur zwei Ziffern, das zur Darstellung von Zahlen verwendet wird. Es wurde bereits Ende des 17. Jahrhunderts von Gottfried Wilhelm Leibniz erfunden. Allerdings erlebt dieses Zahlensystem erst mit der Entwicklung der Digitaltechnik eine starke Verbreitung.

Das Dualsystem ist ein Stellenwertsystem mit der Basis B=2. Die beiden Ziffern mit den Werten null und eins werden gewöhnlich mit den Symbolen 0 und 1 dargestellt. In älterer Literatur findet man im Bezug auf elektronische Datenverarbeitung auch manchmal die Symbole Low (L) und High (H) anstelle von 0 und 1. Somit ist das Dualsystem ein vollständiges Zahlensystem mit der geringstmöglichen Anzahl an verschiedenen Ziffern. Allerdings ergibt sich daraus der Nachteil, dass die Zahlen beim Dualsystem sehr lang sind.

Aufgrund der Tatsache, dass das Dualsystem die Basis B=2 verwendet, hat eine Stelle immer den doppelten Wert der weiter rechts gelegenen Stelle. Dass hierbei die Ziffern 0 und 1 verwendet werden, ist einer der Hauptgründe dafür, dass das

Dualsystem für digitale Rechner geeignet ist, deren elektronische Bauteile ebenfalls binär arbeiten (fließt Strom oder nicht?). So ist beispielsweise der dezimale Wert von 1011 nicht etwa »eintausendundelf«, sondern 11. Diesen Wert erhalten Sie, wenn Sie die Werte in der Zeile »Gesamtwert« addieren:

Ziffern	1	0	1	1
Stellenwert	8	4	2	1
Schema	2^3	2^2	2^1	2^0
Gesamtwert	8	0	2	1

Tabelle A.4 Darstellung einer Dualzahl

Somit ist das Dualsystem im Grunde das einfachste aller Zahlensysteme, da der Stellenwert nie mit einem Ziffernwert multipliziert werden muss. Denn die Stelle ist entweder gesetzt (1), so dass der Stellenwert selbst gilt, oder eben nicht gesetzt (0), dann ist auch der Wert 0.

Dualzahlen finden in der elektronischen Datenverarbeitung bei der Darstellung von Festkommazahlen oder ganzen Zahlen Verwendung. Negative Zahlen werden vor allem als 2-Komplement dargestellt, das nur im positiven Bereich der Dualzahlen-Darstellung entspricht. Um näherungsweise rationale oder gar reelle Zahlen darzustellen, werden vorzugsweise Fließkomma-Darstellungen verwendet, bei der die Zahl normalisiert und in Mantisse und Exponent aufgeteilt wird. Diese beiden Werte werden dann in Form von Dualzahlen gespeichert.

Oktalsystem

Da es recht unbequem ist, Dualzahlen ins Dezimalsystem umzurechnen und umgekehrt, wird gerne das *Oktalsystem* verwendet, mit dem sich Zahlen relativ einfach in Dualzahlen umwandeln lassen und umgekehrt. Das Oktalsystem hat die Basis B=8 – also acht verschiedene Ziffern (0 bis 7). Beim Zählen muss somit beachtet werden, dass nach der 7 nicht die 8, sondern eine Stelle weiter links erhöht werden muss. Somit folgt nach der oktalen 7 die 10. Jede dieser Ziffern einer Oktalzahl kann durch drei Bits dargestellt werden und umgekehrt.

Oktal	0	1	2	3	4	5	6	7
Dual	000	001	010	011	100	101	110	111

Tabelle A.5 Darstellung einer Oktalzahl durch drei Bits

In der Praxis finden Oktalzahlen in der Informatik eigentlich nur noch bei der Darstellung von Dateizugriffsrechten unter Linux/UNIX Verwendung, wo je drei Bits die Rechte einer Benutzerklasse darstellen (siehe Manual-Page von chmod).

Früher waren hierbei auch noch Datenwörter von 24 Bit Länge gebräuchlich, und daher wurden Oktalzahlen zur Eingabe von Bit-Mustern verwendet, weil diese übersichtlicher als Dualzahlen sind und die Umwandlung ins Binärsystem relativ einfach ist. Allerdings beträgt heute die übliche Datenwortlänge 16, 32 und 64 Bit, und dafür ist das Hexadezimalsystem für die Ein- und Ausgabe geeigneter. In der folgenden Tabelle sehen Sie eine systematische Analyse einer Oktalzahl. Der dezimale Wert für die Oktalzahl 1234 beträgt 668. Auch diesen Wert erhalten Sie, wenn Sie die Werte in der Zeile »Gesamtwert« addieren:

Ziffern	1	2	3	4
Stellenwert	512	64	8	1
Schema	8^3	8^2	8^1	8^0
Gesamtwert	512	128	24	4

Tabelle A.6 Darstellung einer Oktalzahl

Hexadezimalsystem

Mit dem *Hexadezimalsystem* (griech.: hexa = sechs, lat.: decem = 10; oder auch seltener *Sezimalsystem* von lat. sedecim = sechzehn) werden Zahlen mit einem Stellenwertsystem der Basis B=16 dargestellt. In der Informatik ist dieses Zahlensystem das beliebteste zur Darstellung von Speicheradressen, Zeichencodes und sonstigen Byte-Inhalten.

Wie beim Oktalsystem lassen sich mit dem Hexadezimalsystem die von der Maschine verbreiteten Zahlen im Dualsystem in einer für den Menschen einfacher lesbaren Weise notieren und verwenden. Neben den gewohnten Symbolen des Dezimalsystems (0 bis 9) werden beim Hexadezimalsystem für die letzten 6 Ziffern die Buchstaben A bis F (oder auch kleingeschrieben a bis f) verwendet. In der folgenden Tabelle finden Sie in der ersten Zeile die hexadezimalen Zahlen, gefolgt von den dualen, den dezimalen und den oktalen Zahlen.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17

Tabelle A.7 Alle wichtigen Zahlensysteme der Informatik

Hinweis [«]

Zwar gibt es für dieses hexadezimale, alphanumerische Mischsystem den Vorschlag, die hexadezimalen Zahlen allgemein mit neuen, unzweideutigen sogenannten *omni-literalen* (d. h. nur Buchstaben) Ziffern darzustellen (siehe dazu: Hexadezimalzeit und hexadezimales Alphabet), aber ob und wann dieses System eingeführt werden soll, kann noch nicht vorhergesagt werden.

Auch hierzu ein Beispiel: Der hexadezimale Wert 2ABC hat im Dezimalsystem den Wert 10940. Diesen Wert erhalten Sie in der folgenden Tabelle, wenn Sie die Werte in der Zeile »Gesamtwert« addieren.

Ziffern	2	A	B	C
Einzelwert	1	10	11	12
Stellenwert	4096	256	16	1
Schema	16^3	16^2	16^1	16^0
Gesamtwert	8192	2560	176	12

Tabelle A.8 Darstellung einer Hexadezimalzahl

Zahlensysteme umrechnen

Zwar gibt es zahlreiche Programme, die das Umrechnen der Zahlensysteme übernehmen, will man aber selbst solche Programme schreiben, sind diese Kenntnisse nötig.

Dezimal nach dual

Dezimalzahlen werden folgendermaßen in Dualzahlen umgerechnet:

Zunächst sucht man die größte in der Zahl vorkommende Zweierpotenz (2^n). Dies kann man entweder durch das fortgesetzte Verdoppeln von 2 ausprobieren, bis die Zahl »passt«, oder aber man hat die Zweierpotenzen im Kopf (1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 usw.). Bis zu einer gewissen Zahl kann man das im Kopf behalten. Bei der Zahl 1200 wäre dies 1024 (2^{10}).

Im nächsten Schritt müssen Sie die entsprechende Zweierpotenz von der Gesamtzahl abziehen und im Ergebnis an vorderster Stelle eine 1 notieren. Von 1200 blieben 176 übrig.

Jetzt überprüfen Sie, ob die nächstniedrigere Zweierpotenz in der Zahl vorkommt. Wenn dies der Fall sein sollte, notieren Sie wieder eine 1 und ziehen diese Zweierpotenz von der Zahl ab. Kommt sie nicht vor, notieren Sie im Ergebnis an vorderster Stelle eine 0. Bei der Zahl 176, die zuvor vom Rest übriggeblie-

ben ist, kommt 512 (2^9) nicht vor, ebenso wie 256 (2^8) nicht darin vorkommt. Also lautet das bisherige binäre Ergebnis 100.

Wenn Sie in dieser Weise für die Reihe der Zweierpotenzen fortfahren bis hinunter zur 1, sieht das Ergebnis für die Zahl 176 folgendermaßen aus:

128 kommt in 176 vor (bisheriges binäres Ergebnis: 1001); der Rest: 48. 64 kommt in 48 nicht vor (binäres Ergebnis somit: 10010). 32 kommt allerdings in 48 vor (binäres Ergebnis: 100101); der Rest: 16. 16 kommt exakt in 16 vor (binäres Ergebnis: 1001011); Rest: 0. Zum Schluss müssen Sie nur noch für die nicht besetzten Stellen 2^3, 2^2, 2^1 und 2^0 eine 0 anhängen (Endergebnis: 10010110000).

Dual nach dezimal

Beim Umrechnen von Dualzahlen in Dezimalzahlen müssen Sie nur die Stellenwerte derjenigen Stellen aufaddieren, die den Wert 1 haben. Am besten fängt man hierbei von rechts bei 2^0 an. Beispiel: 10010110000 soll wieder ins Dezimalsystem umgerechnet werden. Die mit 1 besetzten Stellen sind hierbei 2^4, 2^5, 2^7 und 2^{10}. Daraus ergibt sich folgende Addition:

16 + 32 + 128 + 1024 = 1200

Dezimal nach hexadezimal

Das Umrechnen einer Hexadezimalzahl in eine Dezimalzahl wird folgendermaßen realisiert (als 2345):

Zunächst benötigen Sie die kleinste Sechzehnerpotenz, die größer ist als die umzurechnende Zahl. Damit wissen Sie, dass die höchste besetzte Hexadezimalstelle Ihrer Zahl um einen Wert darunter liegt. Bei der Zahl 2345 wäre die größere Hexadezimalstelle 4096 (16^3). Angefangen wird somit beim Stellenwert 256 (16^2).

Jetzt können Sie die Dezimalzahl durch den soeben ermittelten Stellenwert dividieren. Das ganzzahlige Ergebnis dieser Division ist der gesuchte Ziffernwert, den Sie an der vordersten Stelle notieren können. Den Rest der Division benötigen wir für den nächsten Schritt. Somit wird aus 2345/256=9 Rest 41. Der erste Ziffernwert der Stelle ist also 9. Beachten Sie außerdem, dass bei einem Ergebnis ab 10 mit A bis F geschrieben wird.

Jetzt können Sie den vorherigen Schritt für die restlichen Stellen bis zu 1 (16^0) durchführen. In unserem Fall wäre dies: 41/16=2, Rest 9 (bisherige Hexadezimalzahl: 92). Da es sich beim Rest dieser Berechnung um die Einerstelle handelt, können Sie die 9 als einfachen Ziffernwert hinschreiben. Somit lautet das Endergebnis der Dezimalzahl 2345 umgerechnet als Hexadezimalzahl 929.

Hexadezimal nach dezimal

Die Umrechnung von Hexadezimalzahlen in Dezimalzahlen ist noch einfacher. Sie müssen hierbei nur den jeweiligen Ziffernwert mit dem Wert seiner Stelle multiplizieren und die Ergebnisse addieren. Beispielsweise 12AB:

$12AB = 1*16^3 + 2*16^2 + 10*16^1 + 11*16^0 =$
$1*4096 + 2*256 + 10*16 + 11*1 =$
$4096 + 512 + 160 + 11 = 4779$

Hinweis [«]

Auf die Umrechnung von Oktalzahlen in Dezimalzahlen und umgekehrt wurde nicht eingegangen, weil die Berechnung ähnlich der Umrechnung von Hexadezimalzahlen in Dezimalzahlen und umgekehrt funktioniert.

Oktal nach dual

Das Umrechnen von Oktalzahlen in Dualzahlen ist ebenfalls relativ einfach – was auch der Hauptgrund für die Verwendung von Oktalzahlen ist. Von rechts gesehen entsprechen je drei Dualstellen einer Oktalstelle. Die Umrechnung dieser Dreiergruppen erfolgt daher nach folgendem Schema:

Dual	Oktal	Dual	Oktal
000	0	100	4
001	1	101	5
010	2	110	6
011	3	111	7

Tabelle A.9 Umrechnung von Dualzahlen in Oktalzahlen und umgekehrt

Hexadezimal nach dual

Dasselbe wie bei der Umrechnung von Oktalzahlen in Dualzahlen gilt für Hexadezimalzahlen und Dualzahlen. Hierbei entsprechen je vier Dualstellen von rechts einer Hexadezimalstelle und umgekehrt. Die Umrechnung erfolgt dabei nach folgendem Schema:

Dual	Hexadezimal	Dual	Hexadezimal
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C

Tabelle A.10 Umrechnung von Dualzahlen in Hexadezimalzahlen und umgekehrt

Dual	Hexadezimal	Dual	Hexadezimal
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Tabelle A.10 Umrechnung von Dualzahlen in Hexadezimalzahlen und umgekehrt (Forts.)

Bytes und Maschinenwörter

Da es relativ ineffizient und zudem schwer realisierbar wäre, jedem einzelnen Bit eine eigene Speicheradresse zuzuweisen, werden mehrere von ihnen zu einer Einheit zusammengefasst, die eine gemeinsame Adresse erhält. Diese Speicherstellen, an denen die Daten im Computer gespeichert werden, können durch Nummern angesprochen werden. Will der Prozessor den Inhalt einer Speicheradresse lesen, erhält er jeweils den Wert all dieser Bits; genauso muss er beim Schreiben Werte für alle Bits eines solchen Speicherbereichs liefern.

Die Größe der adressierbaren Speicherstelle hat im Laufe der Computergeschichte variiert, es gab Varianten von vier bis 36 Bits. Dazu gehören auch Varianten, die als »schräg« zu bezeichnen sind, beispielsweise der berühmte PDP-7-Rechner von DEC mit seinen 18 Bits. Auf dem PDP-7 wurde die ursprüngliche Version von UNIX entwickelt.

Die Größe der Speicherblöcke, mit denen ein Computer arbeitet, werden *Maschinenwörter* genannt. Die Anzahl der Bit eines solchen Maschinenworts wird als *Wortbreite* des jeweiligen Prozessors bezeichnet. In den 70er Jahren hat man sich dann darauf geeinigt, dass bei jedem Computer die Adressierung in acht Bit großen Blöcken erfolgt. Diese Blöcke wiederum bezeichnet man als Byte.

[»] **Hinweis**

Gerne wäre ich auf einige Themen etwas detaillierter eingegangen, aber hierbei handelt es sich lediglich um den Anhang. Und außerdem gehört dieses Thema eher zu den Grundlagen eines Informatikers. Wenn Sie Ihr Wissen vertiefen wollen, kann ich Ihnen das Buch »IT-Handbuch für Fachinformatiker« von Sascha Kersken wärmstens empfehlen (erschienen bei Galileo Press), das Sie auf der Buch-CD finden.

A.5 Zeichensätze

Irrtum 1

Um einen Irrtum gleich zu Beginn auszuräumen: Der Standard von C bzw. C++ legt sich nicht auf einen bestimmten Zeichensatz fest.

Was ein Zeichensatz ist, lässt sich relativ einfach erklären. Bei einer Textdatei zum Beispiel handelt es sich um nichts anderes als um eine Abfolge von gleich großen Speicherblöcken, in denen verschiedene Bit-Muster für die unterschiedlichen Zeichen stehen. Wie breit dieser Speicherblock ist, hängt von dem verwendeten Zeichensatz ab. So wird beispielsweise der weitverbreitete ASCII-Zeichensatz mit seinen Erweiterungen mit acht Bit pro Zeichen gespeichert. Unicode hingegen stellt Varianten mit 16 bzw. 32 Bit oder sogar mit variabler Bit-Breite zur Verfügung.

Irrtum 2

Ein häufiger Denkfehler bezieht sich auf den Ausdruck `sizeof(char)`, der immer den Wert `1` zurückliefert. Es wurde bereits im entsprechenden Abschnitt beschrieben, dass `char` immer die Größe von einem Byte hat. Hierbei liegt der Irrtum darin zu glauben, dass die Unicode-Zeichen in einem `char` abgelegt werden. In der Regel ist dies aber nicht der Fall, da Unicode-Zeichen in einem `wchar_t` abgelegt werden. Der Standard schreibt nur vor, dass für ein solches Zeichen ein ganzzahliger Typ verwendet werden muss, dessen Wertebereich ausreicht, um die Zeichen des größten erweiterten Zeichensatzes der Plattform aufzunehmen.

A.5.1 ASCII-Zeichensatz

Der in den 60er Jahren in den USA entwickelte ASCII-Zeichensatz (*American Standard Code for Information Interchange*) war im Grunde nur eine Adaption der amerikanischen Schreibmaschine. Sämtliche Zeichen und Funktionen wurden übernommen, die eine Schreibmaschine erzeugen konnte. Hier findet man auch die typischen Schreibmaschinen-Funktionen wie Zeilenvorschub, Wagenrücklauf oder die Glocke, die das baldige Zeilenende andeutet. Diese sogenannten Steuerzeichen werden von den verschiedenen Terminals, Shells oder Text-Editoren unterschiedlich genutzt.

Der ASCII-Code allein ist nur sieben Bit breit, worin sich also 128 Zeichen (2^7) darstellen lassen. Der verfügbare Platz des achten Bit wurde als erweiterter Zeichensatz genutzt. Aus dem achten Bit wurden verschiedene Zeichensätze entwickelt, die alle die ersten 128 Zeichen gemeinsam haben und die restlichen 128 Zeichen für eigene Erweiterungen nutzen.

Hinweis **[«]**

Auch wenn Unicode stark auf dem Vormarsch ist, bleibt der ASCII-Zeichensatz (ohne Erweiterung) der wichtigste und grundlegendste Computerzeichensatz. Viele (zum Teil ältere) Konsolen- oder Netzwerkanwendungen verarbeiten keine Zeichen über reines ASCII hinaus. Auch Bezeichner der traditionellen Programmiersprachen wie C/C++ dürfen nur aus reinen ASCII-Zeichen bestehen.

In Tabelle A.13 sehen Sie den gesamten ASCII-Zeichensatz. Die Nummerierung ist hier dezimal und hexadezimal. Das Zeichen 'A' zum Beispiel besitzt den hexadezimalen Code 41 oder den dezimalen Code 65. Der Wert der Zeile wird einfach mit dem Wert der Spalte addiert.

Dez		0	16	32	48	64	80	96	112
	Hex	0	10	20	30	40	50	60	70
0	0	NUL	DLE	(Blank)	0	@	P	'	p
1	1	SOH	DC1	!	1	A	Q	a	q
2	2	STX	DC2	»	2	B	R	b	r
3	3	ETX	DC3	#	3	C	S	c	s
4	4	EOT	DC4	$	4	D	T	d	t
5	5	ENQ	NAK	%	5	E	U	e	u
6	6	ACK	SYN	&	6	F	V	f	v
7	7	BEL	ETB	‘	7	G	W	g	w
8	8	BS	CAN	(	8	H	X	h	x
9	9	HT	EM	)	9	I	Y	i	y
10	A	NL	SUB	*	:	J	Z	j	z
11	B	VT	ESC	+	;	K	[	k	{
12	C	NP	FS	,	<	L	\	l	|
13	D	CR	GS	-	=	M	]	m	}
14	E	SO	RS	.	>	N	^	n	~
15	F	SI	US	/	?	O	_	o	(DEL)

Tabelle A.11 ASCII-Code-Tabelle

Die Zeichen 0 bis 31 (hexadezimal 00 bis 1F) sind Steuerzeichen, die mit ihren traditionellen Kurznamen verzeichnet wurden. Beispielsweise ist 0A (dezimal 10) das LineFeed, 0D (13) ist das Carriage Return, und 00 (das Zeichen '\0') wird vor allem als Abschlussmarkierung für C-Strings verwendet.

A.5.2 ASCII-Erweiterungen

Es wurde bereits erwähnt, dass der ASCII-Zeichensatz aus den USA kommt und somit auch nur für englische Texte geeignet ist. Der ASCII-Zeichensatz unterstützt keine Umlaute und sonstigen Sonderzeichen. Da man aber sowieso die einzelnen sieben Bit breiten ASCII-Zeichen in einem acht Bit großen Block speichert, ist in diesem Byte also noch Platz für weitere 128 Zeichen. IBM führte dann einen solchen erweiterten ASCII-Zeichensatz ein, der die wichtigsten diakritischen Zei-

chen der west- und mitteleuropäischen Sprachen und eine Reihe von anderen Grafikzeichen (beispielsweise gerade und gewinkelte Linien) enthält.

Andere Hersteller wiederum erfanden eigene Zeichensätze. Beispielsweise belegte Apple die 128 Zeichen über ASCII anders. Daher können Sie bis heute deutsche Textdateien von einem Mac nur mit Fehlern unter MS Windows und umgekehrt lesen.

Hinweis [«]

MS Windows verwendet übrigens nicht den von IBM erweiterten ASCII-Zeichensatz oder die MS DOS-Variante, sondern den inzwischen standardisierten ANSI-Zeichensatz. Da MS Windows eine grafische Oberfläche hatte, wurden einige Grafikzeichen nicht mehr benötigt. Dieser freigewordene Platz wurde für internationale diakritische Zeichen verwendet. Dies ist übrigens auch der Grund, warum für deutsche Umlaute irgendwelche seltsamen Sonderzeichen angezeigt werden, wenn Sie ein Konsolenprogramm in einem Windows-Editor schreiben.

Leider hatte man nun das Problem, dass die erweiterten Zeichensätze betriebssystemspezifisch waren, da für jede andere Sprache ein anderer Zeichensatz nötig war. Zeitweise hat das niemanden gestört. Die Betriebssystem-Hersteller lösten das Problem, indem sie für jedes Land, in dem das entsprechende System verkauft wurde, einen angepassten Zeichensatz oder eine sogenannte *Codepage* erstellten, um die Besonderheiten der jeweiligen Sprache darstellen zu können.

Zu einem echten Problem wurde dies allerdings, als sich das Internet immer schneller ausbreitete. Daher wurde ein plattformübergreifender und international gültiger Standard für Zeichensätze nötig. Dieser wurde mit der ISO-8559 geschaffen. In der folgenden Tabelle finden Sie die wichtigsten ISO-8859-Zeichensätze.

ISO-Zeichensatz	Alternativer Name	Beschreibung
ISO-8859 – 1	ISO-Latin-1	Entspricht weitgehend dem ANSI-Zeichensatz. Unterstützt die Sprachen Westeuropas und der USA.
ISO-8859 – 2	ISO-Latin-2	Unterstützt zusätzlich einige slawische Sprachen.
ISO-8859 – 3	ISO-Latin-3	Unterstützt Esperanto, Maltesisch und Türkisch.
ISO-8859 – 4	ISO-Latin-4	Dient der Unterstützung der baltischen Sprachen (Estnisch, Lettisch und Litauisch).
ISO-8859 – 5		Kyrillisch
ISO-8859 – 6		Arabisch

Tabelle A.12 Standardisierte ASCII-Erweiterungs-Zeichensätze

ISO-Zeichensatz	Alternativer Name	Beschreibung
ISO-8859 – 7		Griechisch
ISO-8859 – 8		Hebräisch
ISO-8859 – 9	ISO-Latin-5	Wie Latin-1, ersetzt aber isländische durch türkische Zeichen. Wird für Türkisch häufiger verwendet als Latin-3.
ISO-8859 – 10	ISO-Latin-5	Unterstützt Grönländisch und Lappisch.

Tabelle A.12 Standardisierte ASCII-Erweiterungs-Zeichensätze (Forts.)

A.5.3 Unicode

Das Unicode-System wurde eingeführt, um das Problem mit den Schriften zu lösen. Dabei wurde von vornherein mehr Speicherplatz für die einzelnen Zeichen eingeplant, um mehrere unterschiedliche Zeichen darstellen zu können. Die erste Version von Unicode verwendete noch 16 Bit pro Zeichen, um bis zu 65.536 Zeichen darstellen zu können. Seit Unicode 3.1 gibt es auch andere Varianten mit drei bzw. vier Byte, um noch mehr Zeichen in Unicode unterzubringen.

Leider wird Unicode noch nicht von allen Betriebssystemen, Programmiersprachen, APIs und Anwendungen verwendet. Um Probleme zu vermeiden, entsprechen daher die untersten 256 Zeichen exakt dem Zeichensatz von ISO-Latin-1. Mit der Unicode-Codierung UTF-8 können einzelne ASCII-Zeichen sogar in einem Byte dargestellt werden, und für die Darstellung von höheren Zeichen werden dann spezielle Codes verwendet. Damit ist UTF-8 zu ASCII abwärts kompatibel.

Die grundlegende Teilmenge von Unicode (mit den 16 Bit breiten Zeichen) wird als *Basic Multilingual Pane* (kurz BMP) bezeichnet. Dabei handelt es sich um die wichtigsten Zeichen von Sprachen, mathematische Sonderzeichen, Piktogramme, Schmuckelemente und andere Symbole. In der folgenden Tabelle finden Sie eine Übersicht über einige wichtige Codebereiche des BMP-Bereichs von Unicode. Die Codes von Unicode-Zeichen werden üblicherweise im Hexadezimalcode mit vorangestelltem U+ geschrieben; beim BMP-Bereich werden vier Hexadezimalstellen verwendet, ansonsten je nach Bedarf mehr.

Hexadezimal	Dezimal	Codebereich
U+0000 – U+007F	0 – 127	ASCII
U+0080 – U+00FF	128 – 255	ISO-Latin-1
U+0100 – U+017F	256 – 383	Latin Extended A (speziellere diakritische Zeichen)

Tabelle A.13 Codebereiche im Unicode-Block BMP

Hexadezimal	Dezimal	Codebereich
U+0180 – U+024F	384 – 591	Latin Extended B (spezielle, eigentlich nicht lateinische Zeichen, die in einigen Sprachen verwendet werden)
U+0370 – U+03FF	880 – 1023	Griechisch
U+0400 – U+04FF	1024 – 1279	Kyrillisch
U+0590 – U+05FF	1424 – 1535	Hebräisch
U+0600 – U+06FF	1536 – 1791	Arabisch
U+0900 – U+097F	2304 – 2431	Devanagari (Schrift der meisten indischen Sprachen)
U+2200 – U+22FF	8704 – 8959	mathematische Symbole
U+3040 – U+309F	12352 – 12447	Hiragana (japanische Lautschrift)
U+30A0 – U+30FF	12448 – 12543	Katakana (japanische Lautschrift)
U+4E00 – U+9FA5	19968 – 40869	CJK Unified Ideographs (chinesische Zeichen, die zum Teil auch für Japanisch und Koreanisch verwendet werden)

Tabelle A.13 Codebereiche im Unicode-Block BMP (Forts.)

Hinweis **[«]**

Eine Übersicht über alle Zeichenbereiche von Unicode finden Sie auf der Webseite *http://www.unicode.org/*. Dort steht auch ein PDF-Dokument zum Download zur Verfügung, in dem alle einzelnen Zeichen zu sehen sind.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z